FACULTAD DE INFORMATICA
UNLP

TESINA DE LICENCIATURA

TITULO: **Algoritmos paralelos sobre Clusters de Multicores. Análisis de aplicaciones con descomposición funcional y de datos.**

AUTORES: **Fabiana Yael Leibovich**

DIRECTOR: **Dr. Marcelo Naiouf**

CODIRECTOR: **Dra. Laura De Giusti**

CARRERA: **Licenciatura en Sistemas**

Resumen

El objetivo es analizar, investigar y desarrollar algoritmos de *scheduling* para ejecutar eficientemente aplicaciones de procesamiento paralelo sobre arquitecturas *multicore* y *clusters* de *multicore*. Esto implica manejar la distribución de procesos en los *cores* desde la aplicación para obtener ganancia de *performance*. Para ello se utiliza como caso de estudio el problema BASIZ (*Bright and Saturated Image Zones*) que es una aplicación de procesamiento de imágenes para la identificación y detección de las zonas con más brillo e intensidad de color de una imagen que permite implementar soluciones paralelas que admiten tanto paralelismo funcional, paralelismo de datos, como una combinación de los mismos (híbrido). Además, la utilización de BASIZ posibilita el estudio de nuevas técnicas para la programación de algoritmos paralelos que aprovechen eficientemente la potencia de la arquitectura *multicore*, considerando los sistemas híbridos en los que se combina memoria compartida y distribuida.

Las técnicas de asignación de procesos/hilos a procesadores/núcleos incluyen tanto la alternativa de memoria compartida como la de pasaje de mensajes.

Las arquitecturas utilizadas para las pruebas de los algoritmos implementados son *multicore* y *cluster* de *multicore*.

Líneas de Investigación

- Procesamiento paralelo y distribuido
- *Mapping* de procesos/hilos
- Arquitecturas *multicore*
- Librerías de programación paralela

Trabajos Realizados

Se llevó a cabo un análisis de los conceptos fundamentales del procesamiento paralelo para poder así analizar, investigar y evaluar diferentes alternativas para llevar a cabo el *mapping* manual de procesos/hilos a procesadores/núcleos. Entre ellas, jerarquías de memoria, alternativas de *mapping* (memoria compartida y distribuida), librerías de programación paralela. Para ello, se investigó el caso de estudio elegido para luego implementar diferentes soluciones del mismo.

Conclusiones

Luego de realizada la investigación y la experimentación, en función de los resultados obtenidos, puede observarse que utilizando el *mapping* manual (que toma en cuenta las jerarquías de memoria) en soluciones que utilizan diferentes modelos de programación paralela (memoria compartida y pasaje de mensajes) y diferentes estrategias de descomposición (paralelismo de datos, paralelismo funcional y una combinación de los mismos), se mejora la *performance* alcanzable de los mismos si se los compara con la obtenida con el *mapping* del sistema operativo.

Por otro lado, los diferentes experimentos permitieron adquirir un *know-how* importante en cuanto a la asignación de procesos/hilos a procesadores/núcleos. Esto es también escalable en arquitecturas de *cluster* de *multicore* tal como indican los resultados obtenidos.

Trabajos Futuros

Por un lado, la investigación y análisis acerca de la ganancia en *performance* obtenida por el *mapping* manual respecto al llevado a cabo por el sistema operativo, bajo la hipótesis de que se incrementará al aumentar el tamaño del problema.

Por otro lado, implementar un *scheduler* que tenga en cuenta el tipo de aplicación, los recursos del sistema y las jerarquías de memoria existentes, de manera de ser parametrizable al momento de ejecutar una aplicación específica y de la cual el programador de aplicaciones se independice de la necesidad de implementar el *mapping* desde su aplicación. Esto es especialmente de interés si se toma en cuenta que la tendencia es que los procesadores *multicore* no serán todos de propósito general sino que cada núcleo tendrá funciones específicas, tales como multimedia, redes, entre otros.

Fecha de la presentación: Marzo de 2010

Algoritmos paralelos sobre Clusters de Multicores. Análisis de aplicaciones con descomposición funcional y de datos.

Tesina de grado

Alumna: Fabiana Yael Leibovich

Director: Dr. Marcelo Naiouf

Codirector: Dra. Laura De Giusti

Dedicatoria

A mi Papá, mi Mamá y mi Hermano.

A mis abuelos: Fanny, Gregorio y Pablo, que desde algún lugar me están mirando.

Agradecimientos

En primer lugar a mis Padres y a mi Hermano porque sin ellos nada sería posible.

A Papá y Mamá por apoyarme, escucharme, aconsejarme y darme todo y más. Por inculcarme valores y principios, y enseñarme a defenderlos. Por pregonar con el ejemplo. Por enseñarme y mostrarme que pese a todo y pese a muchos, hay que seguir para adelante. Por ser mi ejemplo de vida.

A mi Hermano, por apoyarme y acompañarme siempre.

A mi Director y Codirectora de Tesina, Dr. Marcelo Naiouf y Dra. Laura De Giusti por su apoyo, ayuda, comprensión y humanismo.

A mis Maestros y Profesores que a lo largo de diferentes etapas de mi vida, dejaron su huella en mí; en especial a Armando De Giusti, Cristina Madoz, Marcelo Naiouf, Laura De Giusti, Gladys Gorga, Jorge Runco, Patricia Pesado y Viviana Harari.

A mis amigos, a mis compañeros de Facultad por todos los momentos vividos juntos y a mis compañeros de trabajo.

Por otro lado, agradecerles a todos aquellos que me apoyaron a lo largo de mi Carrera, principalmente a mi Familia, y a las personas que más allá de la relación académica por la cual los conocí me guiaron para definir mi carrera, por enseñarme que de los errores también se aprende, por los consejos, por las charlas y los mates, por los saludos en los pasillos, por estar. Gracias Tito, Cristina, Laura, Marcelo y Jorge por su gran calidad humana.

Gracias a todos aquellos que a su manera estuvieron. A los que están y a los que ya no.

Finalmente, agradezco a la Facultad de Informática y a la Universidad Nacional de La Plata.

Índice de contenidos

Objetivo de la Tesina

El objetivo de esta Tesina es analizar, investigar y desarrollar algoritmos de *scheduling* para ejecutar eficientemente aplicaciones de procesamiento paralelo sobre arquitecturas *multicore* y *clusters* de *multicore*. Esto implica manejar la distribución de procesos en los *cores* desde la aplicación para obtener ganancia de *performance*.

Interesa el estudio de nuevas técnicas para la programación de algoritmos paralelos que aprovechen eficientemente la potencia de la arquitectura, considerando los sistemas híbridos en los que se combina memoria compartida y distribuida.

Para esto, se debe analizar:

a) El problema de la asignación de tareas a núcleos, incluyendo la problemática del balance de carga.
b) La estrategia de descomposición de la aplicación (tanto paralelismo funcional como de datos) y su impacto en la *performance*.
c) La necesidad de combinar memoria compartida y pasaje de mensajes en el uso de *clusters* de *multicore*.

1 Introducción

Reducir el tiempo de ejecución de aplicaciones con grandes requerimientos de procesamiento es el objetivo principal del procesamiento paralelo. Es decir, disminuir los tiempos de ejecución con respecto a los tiempos secuenciales.

Esto se debe al constante incremento del volumen de procesamiento y las limitaciones que impone el cómputo secuencial en cuanto a tiempos de respuesta, acceso a datos distribuidos y manejo de la concurrencia implícita en los problemas del mundo real.

El procesamiento paralelo implica la existencia de múltiples procesadores y se utiliza en una gama muy amplia de aplicaciones entre las

cuales podemos citar aplicaciones científicas, simulaciones y procesamiento de imágenes.

El tipo y organización de los múltiples procesadores influye en el diseño.

A continuación se describirá la arquitectura sobre la cual está centrada esta investigación como así también los fundamentos del procesamiento paralelos que se aplicarán durante el desarrollo de esta Tesina.

1.1 Multicores

Desde el surgimiento del primer procesador, se ha buscado incrementar la capacidad de cómputo y el rendimiento, reduciendo el tamaño físico de los mismos. Para ello es necesario aumentar su potencia de cómputo y al mismo tiempo disminuir el tamaño de los circuitos integrados, lo que conlleva un aumento del nivel de disipación térmica.

Sin embargo, la mejora ilimitada es imposible debido a las limitaciones físicas existentes: el límite de construcción del silicio, material con el cual están construidos los procesadores, ronda los 15-20nm, donde el silicio empieza a ceder por falta de consistencia. Asimismo, el aumento de la frecuencia de trabajo del procesador está limitado por la temperatura que el mismo alcanza y que se hace imposible de disipar. Con el aumento gradual de la temperatura más rápidamente que la velocidad a la que las señales se mueven a través del procesador, conocido como la velocidad de reloj, un aumento en el rendimiento puede crear incluso un mayor incremento en el calor. Esto trajo como consecuencia la necesidad de reevaluar el enfoque que se venía adoptando en el diseño de procesadores. Es así como surgen los procesadores *multicore*.

Un procesador *multicore* consiste en un conjunto de núcleos de procesamiento integrados en un único *chip*.

Normalmente, cada uno de los núcleos (*core*) posee su propio nivel L1 de *cache* y de a pares comparten el nivel L2 de *cache* (incluido generalmente en la placa madre). En la Fig. 1.1 puede verse un ejemplo de una arquitectura *dual core*.

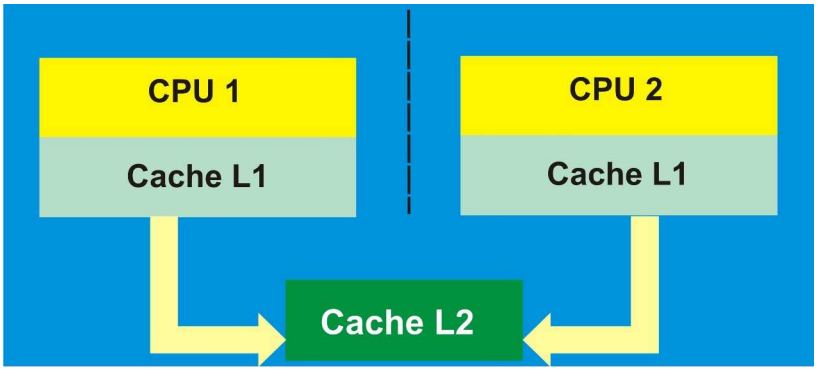

Fig. 1.1 Configuración arquitectura *dual core*.

Los procesadores *multicore* tienen como principal ventaja el aumento de la *performance* sin un incremento significativo del consumo de energía, lo que se traduce en un aumento de *performance* por *watt*.

Las principales características de los procesadores *multicore* son las siguientes:

- Ofrecen una respuesta inmediata y una tecnología rentable para la solución de los desafíos actuales del diseño de procesadores. Proveen un alivio a los problemas derivados del calor y el consumo de energía que existen como consecuencia del continuo avance de frecuencia del monoprocesador, o del incremento de la velocidad del *clock*.
- Ayudan a romper las limitaciones actuales de rendimiento alcanzable que imponen los monoprocesadores y proveen la *performance* necesaria para hacer frente a *software* más complejo que tendrá lugar en un futuro.
- Los sistemas operativos actuales tales como *Microsoft Windows*, *Linux* y *Solaris* son capaces de beneficiarse de los múltiples núcleos.
- Los procesadores *multicore* tienen la potencialidad de ejecutar aplicaciones más eficientemente que los monoprocesadores, brindándoles al usuario la posibilidad de seguir trabajando aunque se estén ejecutando tareas de procesamiento intensivas en *background*, tales como búsquedas en una base de datos, procesamiento de una imagen 3D, descarga de videos de internet, entre otras.
- Aplicaciones de *software* multihilos (programas que ejecutan múltiples tareas o *threads* simultáneamente para aumentar la *performance* en escenarios con gran carga de trabajo), tales como minería de datos,

análisis matemático, y servidores *web*, están bien posicionados para tomar ventaja de las características de los procesadores *multicore*.

Entre los beneficios a largo plazo de las arquitecturas *multicore* se pueden citar:

- La capacidad de ejecutar tanto aplicaciones actuales como así también aplicaciones que se desarrollarán en un futuro, con niveles de complejidad aún mayores que las actuales, lo que significa que el *hardware* mantendrá su vigencia durante un tiempo.
- El crecimiento de la complejidad del *software*, así como el deseo de los usuarios de ejecutar aplicaciones simultáneamente, acelerará la adopción de los procesadores *multicore* en las computadoras personales.
- La próxima generación de aplicaciones requerirá la *performance* que proveen los procesadores *multicore*. El *software* destinado a romper las barreras en la experiencia del usuario, tales como reconocimiento de voz, inteligencia artificial, entre otras, va a ser posible mediante la utilización de arquitecturas *multicore* [1].

1.1.1 Estado actual de la arquitectura *multicore*

El primer procesador multicore introducido en el mercado contaba con dos núcleos o *cores* integrados en un único chip tal como se mostró en la Fig. 1.1.

A nivel mundial, las dos empresas líderes en el desarrollo de procesadores son *Intel* y *AMD*. Ambas empresas comercializan este tipo de arquitecturas y actualmente continúan sus investigaciones y desarrollos.

Intel lanzó su primer procesador *multicore* en el año 2005: *Intel Pentium* D. Fue el comienzo de la tecnología d*ualcore* en microprocesadores, que un año después llevó al desarrollo del Procesador *Intel Core 2 Duo*.

En el año 2007 aparecieron en el mercado los procesadores de cuatro núcleos: *Intel Core 2 Quad*. En 2008, lanzó al mercado los procesadores *Nehalem*. Estos son los procesadores más pequeños del mundo, que utilizan Hafnio como componente, y que se implementaron en los nuevos desarrollos

de *Intel Core 2 Duo*, *Intel Core 2 Quad*, *Intel Xeon* y la línea de reciente aparición, *Intel Atom.* Representan la mayor innovación en la electrónica de los últimos 40 años.

Intel también lanzó al mercado una línea de procesadores que cuenta con 6 núcleos, el *Intel Dunnington*, que forma parte de la línea de procesadores para servidores, además de la nueva línea del 2010: los procesadores *core i3* (2 núcleos), *core i5* (2 núcleos) y *core i7*(de 2 y 4 núcleos) [2].

Recientemente *Intel* hizo público su desarrollo e investigación en la tecnología del futuro: *Terascale Computing*. En sus laboratorios han creado lo que se conoce como *"Single-chip Cloud Computer*", un procesador destinado a la investigación que posee la mayor cantidad de procesadores integrados en un *chip* hasta el momento: 48 núcleos. Éste incorpora la tecnología necesaria para escalar el número de núcleos hasta 100 por *chip*: red interna dentro de cada *chip*, tecnologías avanzadas de gestión de energía, como así también soporte para pasaje de mensajes [3].

Por otro lado, *AMD* también ha desarrollado procesadores *multicore* tales como *AMD Athlon X2*, *AMD Opteron* de 2 núcleos, *AMD Turion X2*, entre otros. También lanzó al mercado procesadores de 3 núcleos (*AMD Phenom X3)* y de 4 núcleos (*AMD Opteron* de 4 núcleos, *AMD Phenom* X4 y *AMD Spider*) [4].

Actualmente *AMD* ofrece dentro de la línea de servidores el *AMD Opteron* de 6 núcleos y anunció que próximamente lanzará al mercado la línea de procesadores *Magny-Cours* que tendrá 12 núcleos [5].

1.1.2 Impacto de la arquitectura *multicore* sobre los sistemas operativos y los lenguajes de programación

La posibilidad de ejecutar múltiples tareas en forma simultánea ha provocado una revolución en el diseño tanto de sistemas operativos como de lenguajes de programación. Estos deben ser capaces de sacar el máximo provecho a esta nueva arquitectura.

Linux ha incluido nuevas técnicas para mejorar su desempeño a través de modificaciones incluidas en el *Kernel 2.6*. Especialmente, las mejoras incluidas para mejorar la escalabilidad del multiprocesamiento simétrico

mejoraron el soporte de procesadores *multicore*. Además, se mejoró el *scheduler* del sistema operativo para dar soporte a arquitecturas *multicore* pero no se lograron los resultados esperados. Especialmente en sistemas de *hyperthreading* y *NUMA*, en los cuales las jerarquías de memoria y la latencia de acceso a la misma no son consideradas por el *scheduler*.

Por último, la presencia de recursos compartidos entre núcleos plantea nuevos desafíos que están siendo estudiados para en un futuro dar mayor soporte desde el *kernel* [6].

En el caso de los sistemas operativos de *Microsoft*, *Windows Vista* y *Windows Seven* soportan la arquitectura *multicore*. Para ello, los desarrolladores de los mismos enfocaron sus esfuerzos en dar soporte a sistemas *NUMA*. Además, mejoraron el *.Net Framework*, el cual dará soporte a partir del *Visual Studio* 2010 para escribir aplicaciones paralelas utilizando la *Task Parallel Library and Parallel LINQ*. Esto permitirá desarrollar aplicaciones que aprovechen las características de la arquitectura *multicore* utilizando para ello lenguajes como *C#* [7][8].

Las nuevas arquitecturas no solo impactan en el desarrollo de sistemas operativos sino también en los lenguajes de programación, que deben incluir características que den soporte a las nuevas configuraciones de *hardware*. Tal es el caso recién mencionado de *Visual Studio* 2010. También se puede nombrar a Java, que a partir de su versión 5.0 da soporte al desarrollo de aplicaciones que utilizan hilos y procesos; el lenguaje C que a través de las librerías *OpenMP* y *Pthreads*, entre otras, posibilita el desarrollo de aplicaciones concurrentes y paralelas. Cabe también mencionar a la librería *MPI*, que permite desarrollar aplicaciones paralelas que utilizan procesos, es decir, que utilizan memoria distribuida. No hay que olvidar que la arquitectura *multicore* permite la programación paralela sobre memoria compartida utilizando mensajes o variables compartidas y posibilita también la programación híbrida combinando ambas técnicas.

1.2 Estrategias de descomposición

El desarrollo de un algoritmo paralelo implica en general un problema complejo. Una tarea importante en el diseño de algoritmos paralelos es la estrategia de descomposición que se utilice en el algoritmo. En este sentido, existen dos estrategias [9]:

Paralelismo de datos: consiste en dividir el volumen de datos del problema en múltiples regiones y asignar esas regiones a diferentes procesos para que sean procesadas. El paralelismo de datos exhibe normalmente una forma natural de escalabilidad. Como ejemplo puede citarse la multiplicación de matrices.

Paralelismo funcional: se trata de identificar las tareas del programa y las dependencias entre ellas para luego planificar la manera en que las tareas que no tienen dependencias puedan ejecutarse en paralelo.

En otras palabras, diferentes procesos llevan a cabo diferentes funciones. Por ejemplo, procesar una secuencia de imágenes sobre las cuales hay que aplicar diferentes filtros. Cada proceso implementará uno de los filtros y cada imagen será procesada por cada uno de los procesos.

1.3 Modelos de algoritmos paralelos

Un modelo de algoritmo paralelo es la forma de estructurar el mismo. Es la combinación del algoritmo mismo con la técnica de descomposición, mapeo y minimización de interacciones que se elija. La utilización de estos modelos permite obtener abstracción sobre el *hardware*, lo que facilita la resolución del problema a resolver.

Existen diferentes modelos de algoritmos paralelos, los que a continuación se detallan [10].

1.3.1 Data-Parallel Model

Es uno de los modelos más simples. Las tareas son mapeadas en forma estática o semi-estática a los procesadores y cada tarea ejecuta las mismas operaciones sobre diferentes conjuntos de datos.

1.3.2 Task Graph Model

En todos los algoritmos paralelos, el procesamiento puede ser visto como un grafo de dependencias de tareas.

En este modelo, se parte del grafo de dependencias para utilizar las interrelaciones de las mismas en la decisión de la localidad de procesos y datos y de esta manera reducir los costos de interacción.

Es típicamente utilizado para resolver problemas en donde el volumen de datos asociado con las tareas es mayor que el tamaño del cómputo asociado a las mismas.

1.3.3 Work Pool Model

El modelo *work pool* o *task pool* está caracterizado por el mapeo dinámico de tareas a procesadores para el balance de carga en donde cada tarea puede ser ejecutada potencialmente por cualquier procesador. No hay un mapeo preestablecido de tareas a procesadores. El mapeo puede ser centralizado o descentralizado.

Normalmente se utiliza este modelo cuando el volumen de datos que debe procesar cada tarea es menor que el procesamiento que debe llevar a cabo la misma. Esto se debe a que las tareas pueden ser reubicadas dinámicamente sin costos altos de interacción de datos (*overhead*).

1.3.4 Master-Slave Model

En este modelo uno o más procesos distribuyen trabajo (*master*) entre los *workers*. El o los procesos *master* saben que tienen que dar trabajo y los procesos *workers* saben que necesitan obtener trabajo del *master*.

La granularidad de las tareas en este modelo debe ser elegida cuidadosamente, teniendo en cuenta que el costo de realizar cómputo debe dominar al costo de la transferencia de trabajo y de sincronización, pero sin generar tareas demasiado grandes que puedan afectar el balance de carga.

1.3.5 Pipeline / Producer-Consumer Model

En el modelo de *pipeline*, un conjunto de datos se pasa a través de una sucesión de procesos, cada uno de los cuales realiza alguna operación sobre ellos. El flujo de nuevos datos provoca que los procesos estén continuamente procesando sobre un conjunto de datos diferente.

El modelo de *pipeline* normalmente utiliza mapeo estático de tareas a procesadores.

1.3.6 Hybrid Model

En algunos casos, más de un modelo puede ser aplicado al problema que se quiere resolver, de esta manera se obtiene un modelo híbrido.

Un modelo híbrido puede estar compuesto tanto por múltiples modelos aplicados jerárquicamente o múltiples modelos aplicados secuencialmente a diferentes fases del algoritmo paralelo.

1.4 Modelos de programación paralela

La elección del modelo de programación que se utiliza afecta la decisión del lenguaje de programación y de la librería a utilizar.

Tradicionalmente el procesamiento paralelo se ha dividido en dos grandes modelos: memoria compartida y pasaje de mensajes [9].

Memoria compartida: todos los datos accedidos por la aplicación se encuentran en una memoria global accesible por todos los procesadores paralelos. Esto significa que cada procesador puede buscar y almacenar datos de cualquier posición de memoria independientemente.

Este modelo se caracteriza por la necesidad de la sincronización para preservar la integridad de las estructuras de datos compartidas.

Pasaje de mensajes: los datos son vistos como si estuvieran asociados a un procesador particular. De esta manera, se necesita de la comunicación entre ellos para acceder a un dato remoto.

Generalmente, para acceder a un dato que se encuentra en una memoria remota, el procesador dueño de ese dato debe enviar el dato y el procesador que lo requiere debe recibirlo. En este modelo, las primitivas de envío y recepción son las encargadas de manejar la sincronización.

Debido al avance de las arquitecturas paralelas y especialmente de la aparición de la arquitectura *multicore*, surge el modelo de programación híbrido en el cual se combinan las estrategias recientemente expuestas. Por ejemplo, en un *cluster* de *multicores*, podría pensarse en utilizar el modelo de memoria compartida para los procesadores lógicos dentro de cada procesador y el modelo de pasaje de mensajes para la comunicación entre los procesadores físicos. Este es un ejemplo de modelo de programación híbrida.

El objetivo de utilizar el modelo híbrido es aprovechar y aplicar las potencialidades de cada una de las estrategias que el modelo brinda, de acuerdo a la necesidad de la aplicación. Esta es un área de investigación de gran interés actual.

1.5 Métricas de *performance*

1.5.1 Tamaño del problema

Se define el tamaño del problema (*W*) como una medida del número total de operaciones básicas necesarias para resolverlo. Dado que puede haber varios algoritmos distintos para resolver el mismo problema, para mantener único el tamaño se lo define como el número de operaciones básicas requeridas por el algoritmo secuencial conocido más rápido en un solo procesador [11][12][13].

1.5.2 Tiempo de ejecución de un algoritmo paralelo

El tiempo de ejecución de un algoritmo paralelo (*Tp*) es el tiempo transcurrido desde el momento en que comienza a ejecutarse el algoritmo paralelo hasta que el último procesador termina su ejecución. Para un sistema

paralelo dado, *Tp* normalmente es una función del tamaño del problema (*W*) y el número de procesadores (*p*) y suele escribirse como *Tp(W,p).*

1.5.3 Speedup

Una de las mediciones de *performance* más usadas en el dominio paralelo intenta describir cuánto más rápido corre la aplicación sobre una máquina paralela. En otras palabras, cual es el beneficio derivado del uso de paralelismo, o cuál es el *speedup* que resulta.

El *speedup* (*S*) es el cociente entre el tiempo de ejecución serial del algoritmo serial conocido más rápido (*Ts*) y el tiempo de ejecución paralelo del algoritmo elegido (*Tp*)[10][14]:

$$S = Ts/Tp$$

Form. 1.1

Los resultados que se obtienen al calcular el *speedup* pueden ser de tres tipos:

- *Speedup* lineal: también llamado *speedup* perfecto debido a que el algoritmo paralelo se ejecuta *p* veces más rápido que el algoritmo secuencial.
- *Speedup* sublineal: es el caso más frecuente en el cual el *speedup* obtenido es menor que *p* debido a factores como comunicación, sincronización, etc.
- *Speedup* superlineal: es el caso menos frecuente, en el cual el *speedup* obtenido supera a *p*. Este caso puede darse cuando por ejemplo el volumen de datos de un programa es tan grande que no entra en la *cache* de un único procesador, entonces cuando ese volumen de datos es dividido para ser procesado por más de un procesador, cada división entra en la *cache* del procesador correspondiente. También suelen encontrarse *speedups* superlineales en problemas de recorrido de

árboles. De esta manera, los tiempos de ejecución pueden presentar una notable mejora que se traduce en una mejora superlineal.

1.5.4 *Overhead* paralelo

El *overhead* paralelo total *To* es la suma de los *overheads* en que incurren todos los procesadores debido al procesamiento paralelo. Incluye los costos de comunicación, trabajo no esencial y tiempo ocioso debido a la sincronización y componentes seriales del algoritmo:

$$To = pTp - Ts$$

Form. 1.2

Asumiendo que *To* es una cantidad no negativa, el *speedup* está acotado por *p*. Para un sistema paralelo dado *To* normalmente es una función de *W* y *p*, por lo que suele denotarse *To(W,p)* [14].

1.5.5 Eficiencia

La eficiencia (E) es una medida de *performance* paralela estrechamente relacionada con el *speedup*, ya que está dada por el cociente entre este (*S*) y el número de procesadores (*p*):

$$E = S/p = Ts / pTp = 1/(1+To/Ts)$$

Form. 1.3

Puede pensarse en la eficiencia como el *speedup* promedio por procesador. Los procesadores no brindan 100 por ciento de su tiempo para cómputo, de modo que la eficiencia mide la fracción de tiempo que son útiles.

El valor de eficiencia se encuentra entre 0 y 1, dependiendo del grado de efectividad con el cual se utilizan los procesadores. Cuando es 1 el *speedup* es perfecto [10][14].

1.5.6 Costo

El costo de un sistema paralelo se define como el producto del tiempo de ejecución paralelo (*Tp*) y el número de procesadores utilizados (*p*). Refleja la suma del tiempo que cada procesador utiliza resolviendo el problema.

Se dice que el sistema paralelo es de costo óptimo si y sólo si el costo es asintóticamente del mismo orden de magnitud que el tiempo de ejecución serial, es decir *pTp = O(W)*. Por lo tanto, el costo de resolver un problema en una máquina paralela es proporcional al tiempo de ejecución del algoritmo secuencial conocido más rápido en un solo procesador. En el caso de que contemos con una arquitectura heterogénea, se debe elegir el procesador de mayor *perfomance* para la ejecución del algoritmo secuencial.

Dado que la eficiencia es el cociente entre el costo secuencial y el costo paralelo, un sistema paralelo de costo óptimo tiene una eficiencia de *O(1)* [10][12][13].

1.5.7 Grado de concurrencia

El grado de concurrencia o grado de paralelismo *C(W)* es el número máximo de tareas que pueden ser ejecutadas simultáneamente en cualquier momento en el algoritmo paralelo. Para un *W* dado, el algoritmo paralelo no puede usar más de *C(W)* procesadores. *C(W)* depende sólo del algoritmo paralelo, y es independiente de la arquitectura. Es una función discreta de tiempo, y refleja cómo el paralelismo de software matchea con el de *hardware*. [10][12][13]

Así definido, el grado de concurrencia supone un número ilimitado de procesadores y otros recursos necesarios disponibles, aunque esto no siempre puede ser alcanzable en una computadora real con recursos limitados.

1.5.8 Ley de Amdahl

Amdahl enunció una ley, que lleva su nombre, en la cual establece que hay una máxima mejora en el rendimiento de un algoritmo que depende de la fracción secuencial del mismo (aquella parte no paralelizable). Esto significa que cuando se alcanza el límite superior de mejora, aunque se agreguen más

procesadores, el *speedup* no aumentará sino que por el contrario se mantendrá constante. Si la parte secuencial de un algoritmo es Sec, existe un límite máximo para el *speedup* alcanzable que está limitado por: $T_{(1)}$/Sec. Es decir, la mejora obtenida en el rendimiento de un sistema debido a la alteración de uno de sus componentes está limitada por la fracción de tiempo que se utiliza dicho componente [15].

1.5.9 Evolución de la Ley de Amdahl: Ley de Gustafson-Barsis

La ley de Gustafson – Barsis, también conocida como corrección de Gustafson-Barsis, está estrechamente relacionada a la ley de Amdahl, a la cual corrige. Establece que si los problemas escalan y el tiempo secuencial crece más que la parte secuencial (no paralelizable) entonces hay un margen mayor para el *speedup*.

Sea *n* una medida del tamaño del problema.

El tiempo de ejecución de un programa en una computadora paralela es descompuesto en:

$$a(n) + b(n) = 1$$

Form. 1.4

Donde:

- *a* es la fracción secuencial
- *b* es la fracción paralela.

En una computadora secuencial, el tiempo relativo será igual a $a(n) + pb(n)$ Donde:

- *p* es el número de procesadores para el caso paralelo

El *speedup* es entonces:

$$(a(n) + pb(n))$$

Form. 1.5

Si la función secuencial *a(n)* disminuye a medida que se incrementa el tamaño del problema *n,* entonces el *speedup* alcanzará *p* cuando se aproxima a infinito.

Por lo tanto la ley de Gustafson rescata el procesamiento paralelo que no era favorecido por la ley de Amdahl [16].

1.5.10 Escalabilidad

Un algoritmo secuencial es evaluado usualmente en términos de su tiempo de ejecución, expresado como una función del tamaño de su entrada. El tiempo de ejecución de un algoritmo paralelo depende no solo del tamaño de su entrada sino también de la arquitectura paralela, el número de procesadores, y características de la máquina tales como: velocidad del procesador, velocidad de los canales de comunicación, topología de interconexión y técnicas de ruteo. Por este motivo no puede evaluarse un algoritmo paralelo aisladamente de la arquitectura paralela sobre la que es implementado. Se denomina sistema paralelo a la combinación de algoritmo y arquitectura paralela sobre la cual corre.

Un algoritmo que posee una buena *performance* para un problema seleccionado sobre un número determinado de procesadores en una máquina dada puede funcionar pobremente si alguno de los parámetros cambia. La escalabilidad de un algoritmo paralelo sobre una arquitectura es una medida de su habilidad para obtener *speedup* creciente linealmente con respecto al número de procesadores, en consecuencia refleja la capacidad del sistema paralelo para usar efectivamente una cantidad creciente de recursos de procesamiento.

Existen diferentes métricas de escalabilidad entre las cuales podemos nombrar Isoeficiencia, fracción serie, entre otros [10].

1.6 Problemas nuevos

La arquitectura *multicore* plantea nuevos desafíos a vencer: el problema de la utilización eficiente de los núcleos de procesamiento.

La posibilidad de contar con múltiples núcleos en un único *chip* abre un panorama nuevo en cuanto a lo que se refiere a programación concurrente y paralela. Esto representa un reto importante a los desarrolladores de aplicaciones paralelas y de sistemas operativos.

¿De qué manera los núcleos existentes en un sistema pueden ser aprovechados eficientemente explotando toda su capacidad?

Si nos remontamos a la ley de Moore, la cual establecía que la cantidad de transistores en un *chip* se duplicaría cada 18 meses, podemos ver que la misma fue cierta mientras no se alcanzaron las limitaciones físicas en cuanto a temperatura y disipación térmica.

Ahora la pregunta que surge: ¿Será esta ley aplicable a la cantidad de núcleos de procesamiento incluidos en un único *chip*?

2 Objetivos

Muchos de los problemas algorítmicos en arquitecturas multiprocesador se han visto fuertemente impactados por el surgimiento de las máquinas *multicore*.

Un *multicore* ofrece a todos los *cores* un acceso rápido a una única memoria compartida, evitando la transferencia de datos entre las máquinas a través de la red; sin embargo, la cantidad de memoria disponible es limitada. Por otra parte, los *clusters* de computadoras tradicionales permiten incrementar el espacio de almacenamiento aunque al precio de alta latencia de la red. Como consecuencia, han surgido sistemas híbridos que permiten combinar las características de ambos (memoria compartida y comunicación por mensajes, lo que además introduce una modificación en la jerarquía de memorias), e incrementar aún más la capacidad y el poder de los sistemas computacionales. Tales sistemas son los *clusters* de *multicores*.

Teniendo en cuenta el auge de la arquitectura de *cluster* de *multicores*, es importante el estudio de nuevas técnicas para la programación de algoritmos paralelos que aprovechen eficientemente la potencia de la arquitectura, considerando los sistemas híbridos en los que se combina memoria compartida y distribuida. Para esto, es necesario analizar:

a) El problema de la asignación de tareas a núcleos.
b) La estrategia de descomposición de la aplicación (tanto paralelismo funcional como de datos) y su impacto en la *performance*.
c) La necesidad de combinar memoria compartida y pasaje de mensajes.

La manera de asignar o mapear procesos lógicos a procesadores físicos es fundamental para la eficiencia de un algoritmo paralelo: el uso desigual (o desbalance) en el uso de los procesadores puede degradar fuertemente la eficiencia del procesamiento paralelo.

Si todos los *cores*/procesadores logran realizar aproximadamente la misma cantidad de operaciones, es posible mejorar el tiempo de respuesta a los requerimientos e incrementar el rendimiento (*throughput*). Pero para ello, es esencial un uso equitativo de los recursos por parte de todos los requerimientos.

Para lograr una planificación adecuada de las operaciones a realizar, es posible utilizar un *scheduler* que sea capaz de predecir el comportamiento de los *cores*/recursos dependiendo del tipo de requerimiento que se esté procesando.

El objetivo de esta Tesina es analizar, investigar y desarrollar algoritmos de *scheduling* para ejecutar eficientemente aplicaciones de procesamiento paralelo sobre arquitecturas *multicore* y *clusters* de *multicore*. Esto implica manejar la distribución de procesos en los *cores* desde la aplicación para obtener mayor ganancia de *performance*.

En relación al segundo ítem a analizar, la forma de descomposición de la aplicación, los problemas de interés son aquellos que admiten como estrategias de descomposición tanto el paralelismo de datos, el paralelismo funcional, como así también una combinación de las mismas. Esto se encuentra estrechamente relacionado a los paradigmas de interacción entre procesos que se utilizarán. Por ello, deben tenerse en cuenta diferentes tipos, tales como: *Master – Worker*, *Pipelining* (paralelismo funcional), y replicación de algoritmos (paralelismo de datos).

Por último, la utilización de *clusters* de *multicore* implica una combinación de los modelos de programación paralela: memoria compartida y

pasaje de mensajes. La comunicación entre procesos que pertenezcan al mismo procesador físico puede pensarse utilizando memoria compartida (nivel micro) mientras que la comunicación entre procesadores físicos (nivel macro) puede pensarse utilizando pasaje de mensajes.

Teniendo en cuenta lo recientemente expuesto, es posible evaluar cómo el *mapping* de procesos a procesadores (*cores*) influye en la *performance* de las aplicaciones teniendo en cuenta las diferentes alternativas de implementación.

3 Características de la arquitectura

Desde el surgimiento de la primera arquitectura paralela, ha habido un gran desarrollo y evolución sobre la misma dando lugar a nuevas arquitecturas. Actualmente, las más importantes son:

a) ***Multicore***: consiste en un conjunto de núcleos de procesamiento integrados en un único *chip*. Normalmente cada uno de ellos posee su propio nivel L1 de *cache* mientras que de a pares comparten el nivel L2 (incluido generalmente en la placa madre).

   Por convención cada uno de los núcleos (*core*) de procesamiento es considerado un procesador lógico mientras que cada procesador (conjunto de *cores* en una placa) es considerado un procesador físico.

b) ***Cluster* tradicional**: se aplica a los conjuntos de computadoras construidos mediante la utilización de componentes de *hardware* comunes y que se comportan como si fuesen una única computadora. Juegan hoy en día un papel importante en la solución de problemas de las ciencias, las ingenierías y del comercio moderno [17].

La tecnología de *clusters* ha evolucionado en apoyo de actividades que van desde aplicaciones de supercómputo y *software* de misiones críticas, servidores *Web* y comercio electrónico, hasta bases de datos de alto rendimiento, entre otros usos.

La utilización de *clusters* para cómputo surge como resultado de la convergencia de varias tendencias actuales que incluyen la disponibilidad de microprocesadores económicos de alto rendimiento y redes de alta velocidad, el desarrollo de herramientas de *software* para cómputo distribuido de alto rendimiento, así como la creciente necesidad de potencia computacional para aplicaciones que la requieran.

Simplemente, un *cluster* es un grupo de múltiples ordenadores unidos mediante una red de alta velocidad, de tal forma que el conjunto es visto como un único ordenador, más potente que los comunes de escritorio. De un *cluster* se espera que presente combinaciones de los siguientes servicios:

1. Alto rendimiento (*High Performance*)
2. Alta disponibilidad (*High Availability*)
3. Equilibrio de carga (*Load Balancing*)
4. Escalabilidad (*Scalability*)

La construcción del *cluster* es fácil y económica debido a su flexibilidad: los ordenadores pueden tener todos la misma configuración de *hardware* y sistema operativo (*cluster* homogéneo), diferente rendimiento pero con arquitecturas y sistemas operativos similares (*cluster* semi-homogéneo), o tener diferente *hardware* y sistema operativo (*cluster* heterogéneo).

Para que un *cluster* funcione como tal, no basta sólo con conectar entre sí los ordenadores, sino que es necesario proveer un sistema de manejo del *cluster*, el cual se encargue de interactuar con el usuario y los procesos que corren en él para optimizar el funcionamiento.

Entre los componentes de un *cluster* se pueden mencionar: los Nodos (ordenadores o servidores), el Sistema Operativo, la Conexión de Red, el *Middleware* (capa de abstracción entre el usuario y los sistemas operativos), los Protocolos de Comunicación y servicios, y por último las aplicaciones (que pueden ser paralelas o no).

c) ***Cluster* de *multicores***: es una arquitectura en la cual existe una colección de procesadores *multicore* interconectados mediante una red, en la que

trabajan cooperativamente juntos como un único recurso de cómputo. Es decir, es similar a un *cluster* tradicional pero con la diferencia de que cada nodo posee un procesador multicore en lugar de un monoprocesador.

3.1 Jerarquías de memoria

La verdadera eficiencia de un algoritmo recae no sólo en la velocidad y cantidad de procesadores sino también en la *performance* del sistema de memoria. Por esto, a la hora de evaluar los diferentes tipos de arquitecturas, se debe tener en cuenta la jerarquía de memoria que posee cada sistema ya que ello incidirá en la *performance* global del mismo.

La *performance* de la jerarquía de memoria está determinada por dos parámetros de *hardware*: latencia de la memoria (tiempo entre que un dato es requerido y está disponible) y el ancho de banda de la misma (la velocidad con la que los datos son enviados de la memoria al procesador). En la Fig. 3.1 puede verse un esquema de la jerarquía de memoria en un sistema monoprocesador.

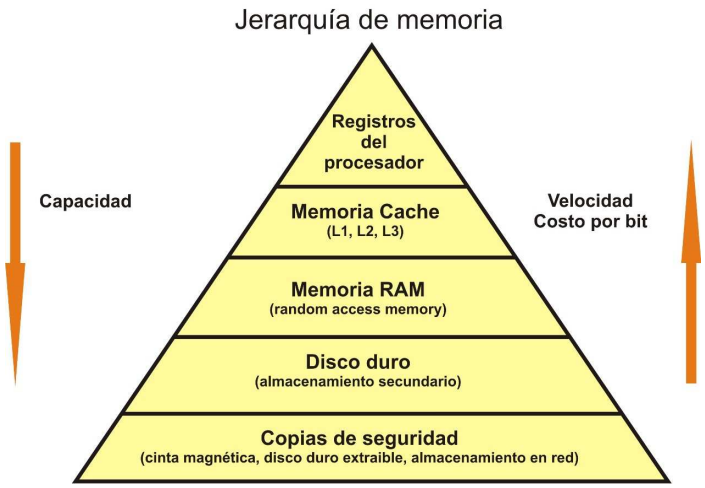

Fig. 3.1

Si se piensa en una arquitectura *multicore* existen, además de los niveles de registros y L1 propio de cada núcleo, dos niveles de memoria: la *cache* compartida de a pares de núcleos (L2) y la memoria compartida entre los *cores* de un procesador *multicore*. Puede verse un ejemplo a continuación en la Fig. 3.2.

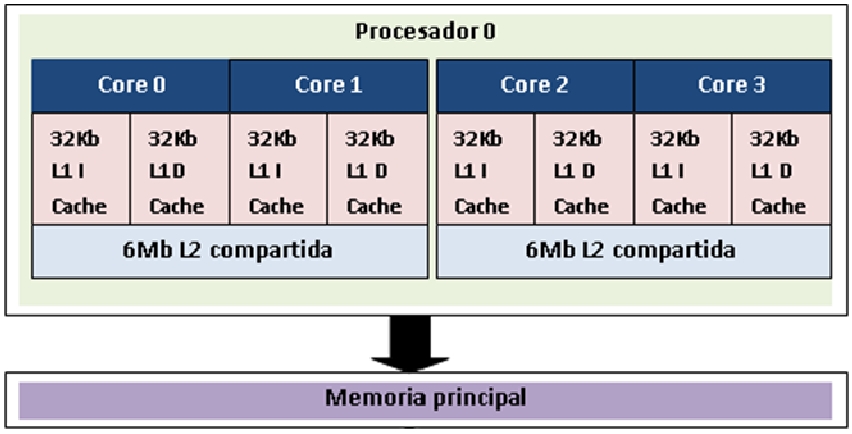

Fig. 3.2

En el caso de *clusters* tradicionales (homogéneos y heterogéneos) existen los niveles propios de cada procesador (registros del procesador y nivel L1 y L2 de *cache*) pero además se incluye un nuevo nivel: memoria distribuida a través de la red, tal como puede verse en la Fig. 3.3.

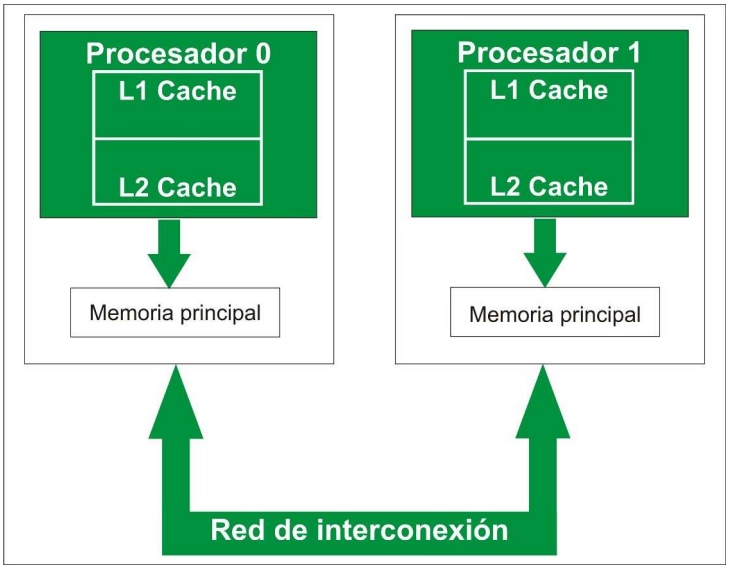

Fig. 3.3

Por último, los *cluster* de *multicore* introducen un nivel más en la jerarquía de memoria. Además de la *cache* compartida entre pares de núcleos, y la memoria compartida entre todos los núcleos de un mismo procesador físico, se agrega la memoria distribuida accesible vía red. La Fig. 3.4 muestra un esquema de esta configuración.

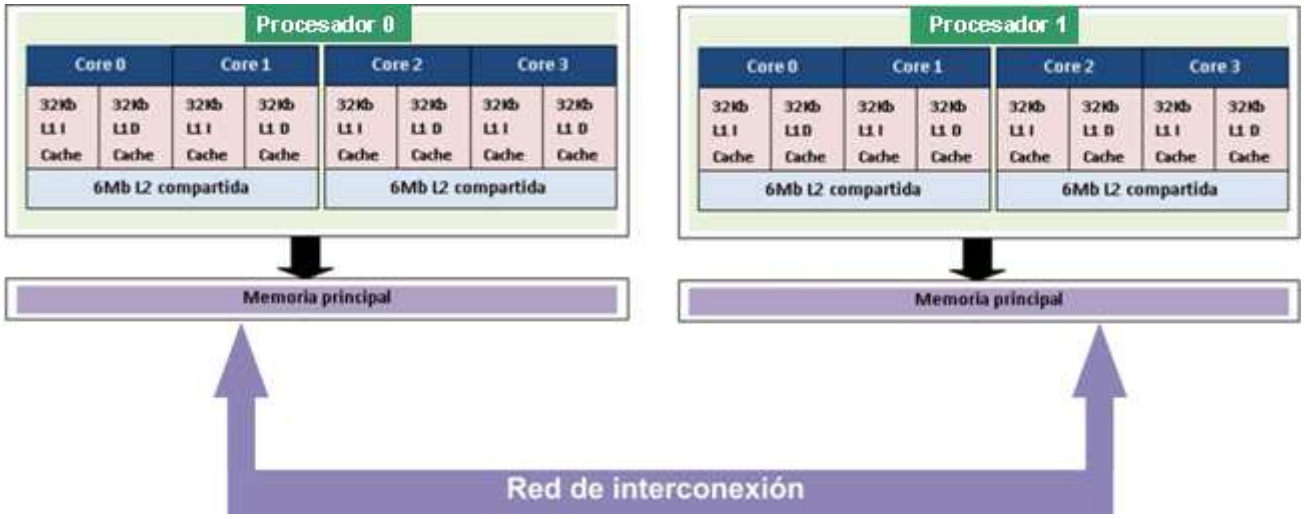

Fig. 3.4

A la hora de implementar un algoritmo paralelo es muy importante considerar la jerarquía de memoria con la que se cuenta, ya que como se mencionó, ello incidirá directamente en la *performance* alcanzable del mismo. Teniendo en cuenta el auge de la arquitectura de *cluster* de *multicores*, los cuales incorporan un nivel nuevo en la jerarquía de memoria, es importante estudiar nuevas técnicas para la programación de algoritmos paralelos que aprovechen eficientemente la potencia de esta arquitectura.

4 Asignación de tareas a núcleos: *mapping* del sistema operativo (*mapping standard*)

Tal como se mencionó en la Sección 2, uno de los objetivos de esta Tesina es mejorar la eficiencia de los algoritmos paralelos realizando un *mapping* manual de los procesos o hilos. Para llegar a ello, es necesario analizar previamente de qué manera el sistema operativo lo implementa. Para ello se han llevado a cabo pruebas en las cuales a través de llamadas al sistema (*system calls*) que provee el sistema operativo *Linux* se analizó la manera en que se realiza el *mapping*.

La llamada al sistema utilizada es *sched_getcpu()* que devuelve el número de *cpu* sobre la que se está ejecutando actualmente el proceso o hilo que lo invoca. Si bien el sistema operativo intenta no cambiar los procesos o hilos de *cpu* debido a los aciertos de *cache* (es decir, tomando en cuenta la

jerarquía de memoria), las pruebas realizadas demuestran que los procesos o hilos cambian continuamente de *cpu* sobre la cual se ejecutan. Esto implica un *overhead* significativo ya que el contexto de los mismos debe migrarse a medida que cambian de núcleo de ejecución.

Por este motivo es que se busca realizar un *mapping* manual que mejore la eficiencia y que se lleve a cabo desde la aplicación misma ya que cada una de ellas tendrá diferentes requerimientos de la arquitectura, y es responsabilidad del programador de la misma establecer donde debe ejecutarse cada tarea.

La tendencia futura es la utilización de procesadores *multicore* que no sólo tengan núcleos de propósito general sino que posean diferentes características, tales como núcleos dedicados al control del video, o a la administración de las redes, entre otros. Esto confirma aún más la necesidad de poder realizar una asignación manual de procesos a núcleos de procesamiento dependiendo del tipo de procesamiento que se lleve a cabo.

5 Métodos de mapeo

La asignación de tareas a núcleos de procesamiento es una fase muy importante en el diseño de algoritmos paralelos. Su objetivo es minimizar el tiempo de ejecución de la aplicación. Existen dos formas de llevarlo a cabo: *scheduling* y *mapping*. En el primer caso, se especifica para cada tarea en qué núcleo y cuándo debe ejecutarse; mientras que en el caso del *mapping* solo se especifica dónde debe ejecutarse. El análisis que se lleva a cabo en este caso se centra en las técnicas de *mapping*.

Según el tipo de sistema que se trate, es decir, memoria compartida o distribuida, se utilizan diferentes técnicas para determinar el mapeo explícito de procesos o hilos a procesadores y/o núcleos. A continuación se detallan las técnicas para cada caso.

5.1 Memoria compartida (alternativa 1)

En los sistemas de memoria compartida no existe una función específica para asignar una tarea a un determinado núcleo de procesamiento. En lugar de ello, existe una función que permite definir la planificación de cada tarea.

La planificación, permite especificar a cuáles de todos los núcleos existentes en un sistema se puede asignar un determinado proceso o hilo. Para poder asignar los mismos a un núcleo particular es necesario establecer que el único procesador/*core* planificable es al que queremos asignarlo.

Los sistemas operativos basados en *Unix* proveen una llamada al sistema que permite definir la afinidad explícitamente. Ésta debe ser utilizada en ambientes con memoria compartida ya que puede administrar los núcleos y procesadores de la máquina sobre la cual está corriendo el sistema operativo. El encabezado de esta función es el siguiente:

*sched_setaffinity(pid_t pid, unsigned long cpusetsize, cpu_set_t *mask)*

Donde:

- ***pid***: es el identificador del proceso al que se le define la afinidad. Si el valor es 0, se asume que es el proceso en ejecución.
- ***cpusetsize***: es la longitud (en *bytes*) de los datos apuntados por el parámetro *mask*.
- ***mask***: representa la máscara de afinidad. Consiste en una máscara de *bits* en la cual cada uno de ellos representa un procesador (lógico) en el sistema. El orden se establece desde el *bit* menos significativo que corresponde al primer procesador lógico hasta el *bit* más significativo que corresponde al último procesador lógico del sistema.

  Si *bit* = 0 ese procesador no será planificable

  Si *bit* = 1 ese procesador será planificable

Por defecto cuando un hilo/proceso se crea es planificable a todos los núcleos existentes en el sistema. Para poder realizar el *mapping* de un proceso

o hilo es necesario establecer que el único procesador planificable es al que queremos asignarlo.

Una de las ventajas que presenta esta técnica de planificación es la posibilidad de modificar dinámicamente (durante la ejecución) la planificación de un determinado proceso. Esto es posible debido a que la llamada del sistema se puede incluir en el código de la aplicación.

Por ejemplo, si se utiliza el lenguaje C, esta llamada puede ser incluida como cualquier otra sentencia del lenguaje. A continuación se muestra un extracto de código en el cual se realiza la paralelización de la multiplicación de dos matrices cuadradas utilizando para ello *Pthreads*.

```
#include <stdio.h>
#include <stdlib.h>
#include <sched.h>
#include <pthread.h>
#include <math.h>

int m, cantidad, cantidadProcesadores;
int *a, *b, *c;

void *multiplicar(void *p){
unsigned long mask;
int i, j, k;
double comienzo, final;

/*Planificación: en función del identificador del hilo y de la cantidad de procesadores del sistema, se lo planifica a un procesador/core particular.*/

int nro = (((int)p+1)%cantidadProcesadores);

mask = pow(2,nro);
if (sched_setaffinity(0, sizeof(mask), &mask) <0) {

                perror("sched_setaffinity");
}
for (i=(int)p*(m/cantidad);i<((int)p*(m/cantidad)+m/cantidad);i++)
  {
      for (j=0;j<m;j++)
      {
          c[i*m+j]=0;

          for (k=0;k<m;k++)
              c[i*m+j]=c[i*m+j]+a[i*m+k]*b[j*m+k];

      }
  }
}

int main (int argc, char *argv[]){
```

```
      if (argc != 4)
            printf("ERROR\n Ingrese los siguientes parámetros: \n Dimensión
de la matriz cuadrada \n Cantidad de Hilos \n Cantidad de procesadores\n");
      else{

            int i;
            double comienzo, final;
            m = atoi(argv[1]);
            cantidad=atoi(argv[2]);
            cantidadProcesadores = atoi(argv[3]);
            a=(int *)malloc(m*m*sizeof(int));
            b=(int *)malloc(m*m*sizeof(int));
            c=(int *)malloc(m*m*sizeof(int));

            int resultadoN;
            pthread_t misThreads[cantidad];
            inicializarMatrizPorFila(a, m);
            inicializarMatrizPorColumna(b, m);
            /*Crear los Threads*/
            for (i=0; i<cantidad; i++){
                  resultadoN    =    pthread_create(&misThreads[i],    NULL,
multiplicar,  (void*)i);
                  if(resultadoN)
                  {
                        printf("ERROR      creando      thread      código:
%d",resultadoN);
                        exit(-1);
                  }
            }
            for (i=0; i<cantidad; i++){
                  pthread_join(misThreads[i], NULL);

            }

            pthread_exit(NULL);
      }
return 0;
}
```

Una de las desventajas que presenta esta alternativa es que para poder llevar a cabo la planificación, el código fuente de la aplicación debe ser modificado. Por otro lado, al ser una planificación dinámica e incluida dentro del código fuente, consumirá tiempo de *cpu*, lo que incidirá en el tiempo de ejecución final.

5.2 Memoria distribuida (Pasaje de mensajes)(alternativa 2)

A la hora de utilizar librerías de comunicación para sistemas distribuidos existen diferentes alternativas, tales como: *PVM* y *MPI*, entre otras. En la actualidad, la librería más utilizada es *MPI.*

Open MPI es un proyecto de código abierto que implementa el *standard* de *MPI* y que como funcionalidad extra al mismo provee directivas para la asignación explícita de procesos a núcleos de procesamiento. Para ello requiere de dos archivos llamados *rankfile* y *hostfile*.

En el archivo *hostfile* se define el número de núcleos disponibles en el sistema y el nombre de la máquina dentro de la red. El formato del archivo es el siguiente:

- ***hostNameX slots = nro de núcleos***
- ***hostNameY slots = nro de núcleos***

En el archivo ***rankfile*** se define una entrada por cada proceso de la siguiente manera:

- ***rank N = hostNameX slot = nro de cpu***
- ***rank M = hostNameY slot = nroSocket:nroNúcleo***

La Fig. 5.1 es un esquema en el que se especifica qué representan los parámetros nro de cpu, nroSocket y nroNúcleo.

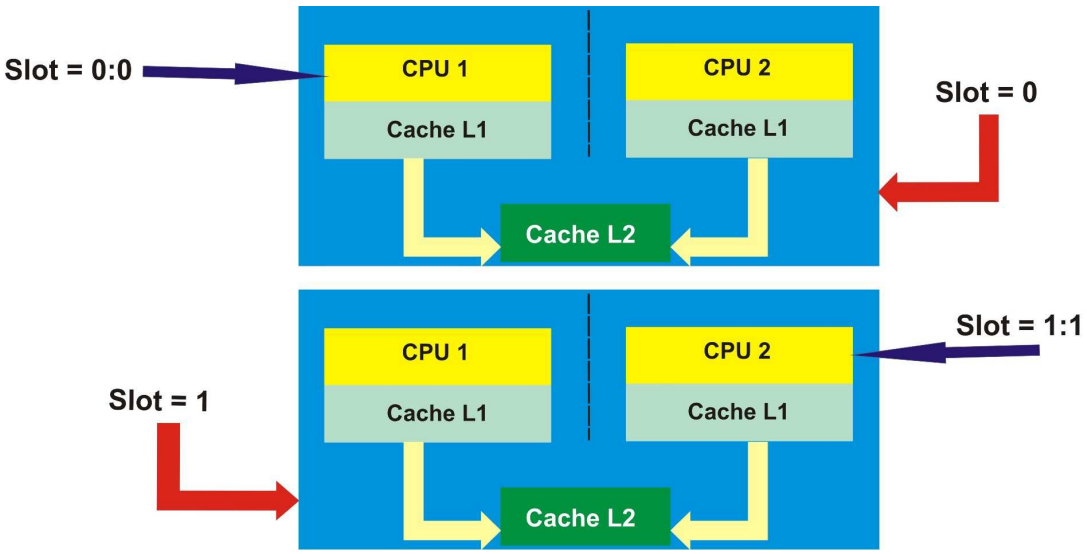

Fig. 5.1

Estos dos archivos deben ser pasados como parámetro en el momento de la ejecución de la siguiente manera:

mpirun -np *nro_de_procesos_a_crear* ***-hostfile*** *archivo_hostfile* ***–mca rmaps_rank_file_path*** *archivo_rankfile ejecutable parametros*

La ventaja de esta alternativa es que el código fuente queda intacto, no es necesario modificarlo para agregar esta funcionalidad y como consecuencia el *mapping* no consume tiempo de ejecución. La desventaja es que la asignación es estática, ya que como es pasada como parámetro en el momento de la ejecución, no se puede modificar dinámicamente.

En función de la arquitectura que se analiza en esta Tesina, es decir, *cluster* de *multicore*, puede pensarse en utilizar ambas alternativas de mapeo combinándolas entre sí para obtener una mejor *performance* global del sistema. Esto es aplicable siempre y cuando el algoritmo que se analice utilice un modelo hibrido, combinando memoria compartida con pasaje de mensajes. Tal es el caso de las soluciones al problema BASIZ presentadas en las secciones 7.2.4.1 y 7.2.4.2.

5.3 Mapeo en *cluster* de *multicore*

El mapeo en la arquitectura de *cluster* de *multicore* (explicada en la Sección 3), debe tener en cuenta las dos alternativas recién planteadas dada la jerarquía de memoria que posee.

Si se piensa en un algoritmo que esté dividido en dos niveles jerárquicos, el primer nivel será de división a nivel de procesos para que sean mapeados a los procesadores dentro del *cluster*. El segundo nivel será el de hilos dentro de cada uno de los procesos creados. De esta manera se obtiene un algoritmo híbrido que puede aprovechar las potencialidades de ambas técnicas (memoria compartida y pasaje de mensajes).

El mapeo de los procesos se llevará a cabo utilizando la alternativa 2 que provee la librería de pasajes de mensajes *Open MPI*, mientras que para la planificación de los hilos se utilizará la alternativa 1.

6 Librerías de programación paralela

Las librerías de programación paralela que actualmente más se utilizan son *Pthreads* y *OpenMP* en el caso de memoria compartida y *OpenMPI* para pasaje de mensajes. Estas librerías son las que se usan en el trabajo experimental de esta Tesina. Por este motivo a continuación se explican y ejemplifican cada una de ellas.

6.1 Pthreads

A lo largo del tiempo, los fabricantes de *hardware* han implementado sus propias versiones para el manejo y administración de hilos. Estas versiones difieren mucho unas de otras, dificultando a los programadores desarrollar aplicaciones multihilos que sean portables. Por este motivo, en el año 1995 la IEEE estableció el *standard POSIX* (***P****ortable* ***O****perating* ***S****ystem* ***I****nterface*). La última versión que existe es la IEEE *Std* 1003.1, 2004 *Edition*.

Pthreads es una librería que implementa el *standard POSIX y* está compuesto por un conjunto de tipos y llamadas a procedimientos en el lenguaje de programación C que incluye un *header file* y una librería de hilos que forma parte por ejemplo de la librería *libc*, entre otras. Se utiliza para la programación de aplicaciones paralelas que utilizan memoria compartida [18].

Las subrutinas que conforman la *API* de *Pthreads* pueden ser clasificadas en cuatro grandes grupos:

- Manejo de hilos: rutinas que trabajan directamente con los *threads* - crear, separar, unir, etc. También incluyen funciones para establecer/consultar atributos de hilo (acoplables, *scheduling*, etc.).
- *Mutex*: rutinas que se ocupan de la sincronización y exclusión mutua. *Mutex* (semáforos) proporciona funciones para crear, destruir, bloquear y desbloquear semáforos. Estos se complementan con las funciones de atributos de semáforos que establecen o modifican los atributos asociados con los mismos.
- Variables condición: rutinas para el manejo de la sincronización por condición entre hilos que comparten semáforos. Provee funciones para

crear, destruir, *wait* y *signal* sobre las variables. También provee funciones para establecer/consultar atributos de las mismas.

- Sincronización: rutinas que manejan *locks* y barreras.

A continuación se muestra un ejemplo en el cual se realiza la paralelilzación de la multiplicación de matrices cuadradas utilizando para ello *Pthreads*.

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <sched.h>
#include <pthread.h>
#include <mpi.h>
#include <math.h>


void multiplicarMatricesFilCol(int *a, int *b, int *c, int m, int cantF);
void inicializarMatrizPorFila(int *mat, int n);
void inicializarMatrizPorColumna(int *mat, int n);
int m, cantidad, cantidadProcesadores;
int *a, *b, *c;


void *multiplicar(void *p){
unsigned long mask;
int i, j, k;
for (i=(int)p*(m/cantidad);i<((int)p*(m/cantidad)+m/cantidad);i++)
  {
      for (j=0;j<m;j++)
      {
          c[i*m+j]=0;

          for (k=0;k<m;k++)
              c[i*m+j]=c[i*m+j]+a[i*m+k]*b[j*m+k];

      }
  }
}

int main (int argc, char *argv[]){
      if (argc != 4)
            printf("ERROR\n Ingrese los siguientes parámetros: \n Dimensión
de la matriz cuadrada \n Cantidad de Hilos \n Cantidad de procesadores\n");
      else{

            int i;
            double comienzo, final;
            m = atoi(argv[1]);
            cantidad=atoi(argv[2]);
            cantidadProcesadores = atoi(argv[3]);
            a=(int *)malloc(m*m*sizeof(int));
            b=(int *)malloc(m*m*sizeof(int));
            c=(int *)malloc(m*m*sizeof(int));
            int resultadoN;
            pthread_t misThreads[cantidad];
            inicializarMatrizPorFila(a, m);
```

```
            inicializarMatrizPorColumna(b, m);
            /*Crear los Threads*/
            comienzo = MPI_Wtime();
            for (i=0; i<cantidad; i++){
                  resultadoN = pthread_create(&misThreads[i], NULL,
multiplicar,  (void*)i);
                  if(resultadoN)
                  {
                        printf("ERROR creando thread código:
%d",resultadoN);
                        exit(-1);
                  }
            }
            for (i=0; i<cantidad; i++){
                  pthread_join(misThreads[i], NULL);

            }
            final = MPI_Wtime();
            printf( "Número de segundos transcurridos del programa: %f
s\n",final - comienzo);

            pthread_exit(NULL);
      }
return 0;
}


void inicializarMatrizPorFila(int *mat, int n){

      int i, j, valor;      /* Indices */

      for (i = 0; i < n; i++){
            for (j = 0; j < n; j++){
                  valor = (int) rand() % 20;
                  if (valor > 10) {
                        valor -= 10;
                        valor*= -1;
                  }
                  mat[i*n +j] = 1;
            }
      }
}

void inicializarMatrizPorColumna(int *mat, int n){

      int i, j, valor;      /* Indices */

      for (i = 0; i < n; i++){
            for (j = 0; j < n; j++){
                  valor = (int) rand() % 20;
                  if (valor > 10) {
                        valor -= 10;
                        valor*= -1;
                  }
                  mat[(j * n) + i] = 1;
            }
      }
}
```

6.2 OpenMP

OpenMP es una interfase de programación (*API*) definida por un conjunto de fabricantes de *hardware* y *software* entre los que se encuentran: *Sun Microsystems*, *IBM, Intel, AMD*, entre otros.

Provee de una interfase de programación portable y escalable para desarrolladores de aplicaciones paralelas que utilizan memoria compartida. Esta *API* soporta C/C++ y *Fortran* en múltiples arquitecturas, incluyendo *LINUX* y *Windows NT*. Provee varios constructores y directivas para especificar regiones paralelas, trabajo compartido, sincronización y variables de entorno [19].

Está constituído por los siguientes tres componentes:

- Directivas de compilación
- Biblioteca de rutinas en tiempo de ejecución
- Variables de entorno

Una de las ventajas que presenta *OpenMP* por sobre *Pthreads* es que los hilos se encuentran creados durante toda la ejecución del programa; al encontrar una zona paralela los hilos se activan y comienzan a ejecutarse hasta el fin de la zona paralela pero al estar ya creados, no hay consumo ni de recursos ni de tiempo de ejecución para crear y destruir los mismos. En *Pthreads* los hilos se crean y se destruyen cada vez que se utilizan. Sin embargo, la ventaja que posee *Pthreads* por sobre *OpenMP* es que el programador tiene mayor control sobre la creación, destrucción y comportamiento del hilo ya que el manejo de los mismos es a más bajo nivel que en *OpenMP*.

A continuación se muestra un ejemplo en el cual se realiza la paralelilzación de la multiplicación de matrices cuadradas utilizando para ello *OpenMP*.

```
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>

void inicializarMatrizPorFila(int *mat, int n);
void inicializarMatrizPorColumna(int *mat, int n);
int m, cantidad;
```

```
int *a, *b, *c;

main (int argc, char *argv[])
{ int nthreads, tid;
      int i,j,k;
      double comienzo, final;
      m = atoi(argv[1]);
      cantidad=atoi(argv[2]);
      a=(int *)malloc(m*m*sizeof(int));
      b=(int *)malloc(m*m*sizeof(int));
      c=(int *)malloc(m*m*sizeof(int));
        inicializarMatrizPorFila(a, m);
      inicializarMatrizPorColumna(b, m);
      omp_set_num_threads(cantidad);
      int sum = omp_get_thread_num();
      comienzo = MPI_Wtime();

#pragma omp parallel private(i,j,k) shared (a, b, c, m)
   {  #pragma omp for schedule(static)
      for (i = 0; i < m ; i++){
            for (j = 0; j < m ; j++)
                  {
                  c[i*m +j] = 0;
                        for (k = 0; k < m ; k++)
                        c[i*m+j]=c[i*m+j]+a[i*m+k]*b[j*m+k];

                  }
      }
   }

final = MPI_Wtime();
printf( "Número de segundos transcurridos del programa: %f s\n",final -
comienzo);
  }


void inicializarMatrizPorFila(int *mat, int n){

      int i, j, valor;      /* Indices */

      for (i = 0; i < n; i++){
            for (j = 0; j < n; j++){
                  valor = (int) rand() % 20;
                  if (valor > 10) {
                        valor -= 10;
                        valor*= -1;
                  }
                  mat[i*n +j] = 1;
            }
      }
}

void inicializarMatrizPorColumna(int *mat, int n){

      int i, j, valor;      /* Indices */

      for (i = 0; i < n; i++){
            for (j = 0; j < n; j++){
                  valor = (int) rand() % 20;
                  if (valor > 10) {
```

```
                        valor -= 10;
                        valor*= -1;
                    }
                    mat[(j * n) + i] = 1;
                }
            }
}
```

6.3 Open MPI

MPI es una interfase de pasaje de mensajes creada para resolver el inconveniente que surgió luego de que cada fabricante de *hardware* definiera su propia interfase de comunicación, generalmente incompatible con las demás. El objetivo de *MPI* es solucionar este problema definiendo para ello un *standard.*

MPI es una librería que puede ser utilizada para desarrollar programas que utilizan pasaje de mensajes (memoria distribuida) utilizando para ello los lenguajes de programación C o *Fortran*. El *standard MPI* define tanto la sintaxis como la semántica del conjunto de rutinas que pueden ser utilizadas en la implementación de programas que utilicen pasaje de mensajes. Brinda más de 125 rutinas, pero posee unas pocas claves conceptuales las que facilitan notablemente su utilización [10].

Una de las implementaciones de este *standard* es *OpenMPI* que es la que se utiliza en las implementaciones realizadas en esta Tesina [12].

A continuación se muestra un ejemplo en el cual se realiza la paralelización de la multiplicación de matrices cuadradas utilizando para ello *OpenMPI*.

```
#include <stdio.h>
#include<stdlib.h>
#include <mpi.h>
#include <malloc.h>
#include<sched.h>

/* Inicializar matriz con valores enteros aleatorios */
void inicializar_matriz(int *mat, int n);

/* Multiplicar matrices cuadradas */
void multiplicarMatricesFilCol(int *a, int *b, int *c, int m, int
cantF);
void inicializarMatrizPorFila(int *mat, int n);
void inicializarMatrizPorColumna(int *mat, int n);

int main (int argc, char *argv[]) {
```

```
      MPI_Init(&argc, &argv);
      int n = atoi(argv[1]);
      int cantProc, rank;
      int *a;
      int *b;
      int *c;

      a = (int*) malloc(n * n * sizeof(int));
      b = (int*) malloc(n * n * sizeof(int));
      c = (int*) malloc(n * n * sizeof(int));
      unsigned long mask;
      int filas_matriz;
      double tiempoInicio, tiempoFin;
      int er;


      /*Inicializar MPI*/

      MPI_Comm_size(MPI_COMM_WORLD, &cantProc);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      er=n*n;
      /*Dividir la cantidad de Filas por la cantidad de Procesos y
multiplicar por n porque es la cantidad de elementos a enviar*/
      filas_matriz = (n / cantProc) * n;

     if(rank==0){
            double *tiempos;
            double tiempoInicioPrograma;
            tiempos = (double*) malloc(cantProc* sizeof(double));
            inicializarMatrizPorFila(a, n);
            inicializarMatrizPorColumna(b, n);

            tiempoInicioPrograma = MPI_Wtime();
            MPI_Scatter(a, filas_matriz, MPI_INT, a, filas_matriz,
MPI_INT, 0, MPI_COMM_WORLD);
            MPI_Bcast(b, er, MPI_INT, 0, MPI_COMM_WORLD);

            tiempoInicio = MPI_Wtime();
            multiplicarMatricesFilCol(a, b, c, n, n/cantProc);
            MPI_Gather(c, filas_matriz, MPI_INT, c, filas_matriz,
MPI_INT, 0, MPI_COMM_WORLD);
            tiempoFin = MPI_Wtime();
            double tiempTot=tiempoFin-tiempoInicio;

            MPI_Gather(&tiempTot, 1, MPI_DOUBLE, tiempos, 1,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
            int i;
            printf("Tiempo total del programa: %f s\n",MPI_Wtime()-
tiempoInicioPrograma);

            for(i=0;i<cantProc;i++){
                  printf( "Total: %f Proceso: %d s\n", tiempos[i], i);
            }

      }
      else {
            MPI_Scatter(a, filas_matriz, MPI_INT, a, filas_matriz,
MPI_INT, 0, MPI_COMM_WORLD);
            MPI_Bcast(b, er, MPI_INT, 0, MPI_COMM_WORLD);
```

```
            tiempoInicio = MPI_Wtime();
            multiplicarMatricesFilCol(a, b, c, n, n/cantProc);
            tiempoFin = MPI_Wtime();

            MPI_Gather(c, filas_matriz, MPI_INT, c, filas_matriz,
MPI_INT, 0, MPI_COMM_WORLD);
            double tiempTot=tiempoFin-tiempoInicio;
            MPI_Gather(&tiempTot, 1, MPI_DOUBLE, c, 1, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

      }
MPI_Finalize();
return 0;
}
void inicializarMatrizPorColumna(int *mat, int n){

      int i, j, valor;

      for (i = 0; i < n; i++){
            for (j = 0; j < n; j++){
                  valor = (int) rand() % 20;
                  if (valor > 10) {
                        valor -= 10;
                        valor*= -1;
                  }
                  mat[(j * n) + i] = valor;
            }
      }
}
void inicializarMatrizPorFila(int *mat, int n){

      int i, j, valor;

      for (i = 0; i < n; i++){
            for (j = 0; j < n; j++){
                  valor = (int) rand() % 20;
                  if (valor > 10) {
                        valor -= 10;
                        valor*= -1;
                  }
                  mat[i*n +j] = valor;
            }
      }
}
void multiplicarMatricesFilCol(int *a, int *b, int *c, int m, int
cantF){
      int i, j, k;

  for (i=0;i<cantF;i++)
  {
      for (j=0;j<m;j++)
      {
          c[i*m+j]=0;

          for (k=0;k<m;k++)
              c[i*m+j]+=a[i*m+k]*b[j*m+k];
      }
  }
}
```

6.4 Ct: C for Throughput Computing

Uno de los principales retos para poder migrar aplicaciones *multicore* en el futuro es poder migrar las herramientas que utilizan los programadores, ambientes de compilación, y millones de líneas de código a nuevos modelos y compiladores de programación paralela. Para ayudar en esa transición, investigadores de *Intel* están desarrollando *Ct*, o también llamado *C/C++ for Throughput Computing* [21].

Ct trabajará con cualquier compilador *C++ standard* porque es una librería de compilación *C++ standard* (con muchas *runtimes* por detrás). Cuando se inicializa la librería *Ct*, se carga un *runtime* que incluye el compilador, *threading runtime*, manejador de memoria, esencialmente todos los componentes necesarios para generar código que utilice hilos o vectores. El código *Ct* es compilado dinámicamente, de manera que el *runtime* intenta agregar la mínima cantidad de tareas o trabajo de paralelismo de datos para poder minimizar el *overhead* de los hilos y controlar la granularidad de acuerdo a las condiciones en que se ejecutará ese código. El principal objetivo de los desarrolladores de esta librería es permitir generar código que sea fácilmente migrable a nuevas arquitecturas *multicore*. En la Fig. 6.1 se muestra un gráfico en el cual se explican estos conceptos.

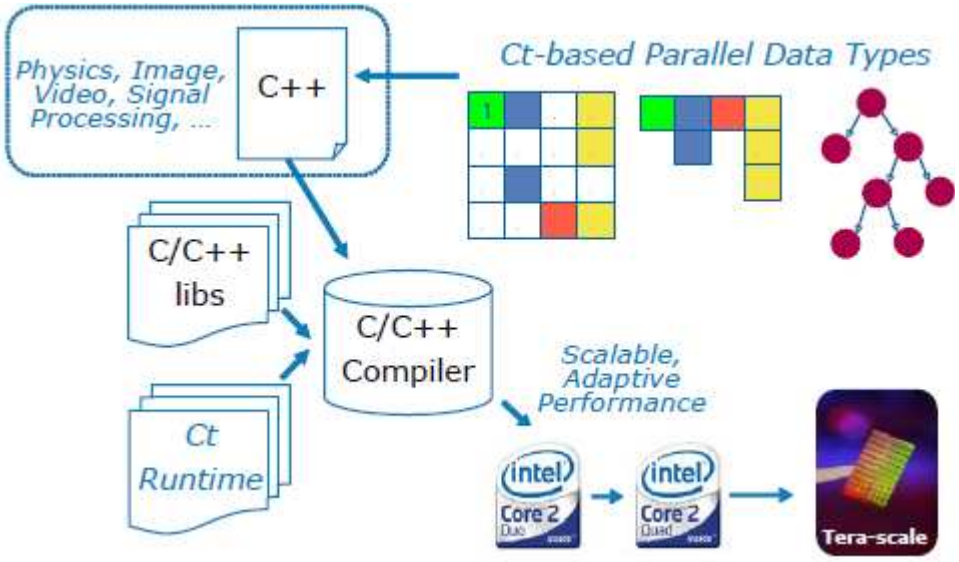

Fig. 6.1

7 Trabajo experimental

Para poder llevar a cabo el análisis y la investigación planteada se elige como caso de estudio una aplicación de procesamiento digital de imágenes. Esta aplicación tiene la particularidad de admitir soluciones que utilizan como modelos de paralelismo tanto el paralelismo funcional como el de datos, y como consecuencia admite diferentes paradigmas de interacción entre procesos. Esta característica distintiva permite realizar experimentación en una variada gama de alternativas, posibilitando el análisis del comportamiento de la *performance* del sistema, teniendo en cuenta los niveles de análisis ya planteados.

7.1 Caso de estudio

Como caso de estudio se eligió BASIZ (*Bright and Saturated Image Zones*) que es una aplicación de procesamiento de imágenes para la identificación y detección de las zonas con más brillo e intensidad de color de una imagen. Esta aplicación es comúnmente utilizada cuando se desea analizar particularmente el paralelismo de datos, el paralelismo funcional, como así también una combinación de ambos, con grandes volúmenes de procesamiento de información [22][23]. El procesamiento digital de imágenes es frecuentemente utilizado en áreas tales como el procesamiento de imágenes médicas, satelitales e investigación astronómica.

Dada una imagen de entrada, el algoritmo lleva a cabo una serie de fases de procesamiento obteniendo como resultado la imagen original en la cual están marcadas las zonas más perceptibles al ojo humano, que son las zonas más brillantes [22][23].

Las fases de procesamiento de la aplicación están compuestas por siete pasos lógicos que pueden ser divididos en subtareas. Estos 7 pasos son [24]:

1. **Separación** en los tres canales de color básicos (rojo, verde y azul).
2. **Difuminado** de cada uno de los tres canales de color básicos (*bluring*).
3. **Suma** de las imágenes difuminadas para cada canal de color.
4. **Unión** de los tres canales de color difuminados.

5. **Conversión** de formato de representación de la imagen difuminada de *RGB* a *HSV* (matiz, saturación y brillo).
6. **Umbralización**: obtener el umbral a partir del cual se determina la inclusión de los *pixels* a un segmento sensitivo (*thresholding*).
7. **Detección** y marcado de las zonas más sensitivas al ojo humano.

7.2 Soluciones implementadas

Los estudios experimentales fueron realizados en base a la implementación del algoritmo en diferentes versiones. Para ello se utilizó el lenguaje C como lenguaje de implementación y no se utilizaron librerías destinadas al procesamiento digital de imágenes debido a que el encapsulamiento de las funciones que proveen, imposibilita una paralelización de ese código.

Con el objetivo de realizar el análisis de la *performance* alcanzable contrastando el *mapping* manual y el automático (llevado a cabo por el sistema operativo) se implementó una solución secuencial y diferentes variantes de algoritmos paralelos. En cada una de ellas se varía el modelo de algoritmo paralelo (paralelismo funcional, de datos, o una mezcla de ambos) y el paradigma de interacción entre procesos (*Pipeline* y *Master-Worker*).

A continuación se describen cada una de las soluciones implementadas.

7.2.1 Solución secuencial

{Para cada una de las imágenes}

1. *Separar la imagen en los tres canales de color (rojo, verde y azul) y generar una imagen para cada uno de ellos.*
2. *Aplicar el difuminado a los tres canales de color. Para cada canal se aplica un filtro gaussiano de 5X5 en tres niveles, generando una imagen para cada nivel.*
3. *Para cada canal de color, sumar las tres imágenes difuminadas generando una nueva imagen.*

4. *Unir las tres imágenes difuminadas generando una nueva imagen.*
5. *Convertir el formato de representación de la imagen recién generada de RGB a HSV (matiz, saturación y brillo).*
6. *Generar el umbral a partir del cual se determinará si un pixel pertenece o no a una zona sensitiva. Luego generar una matriz binaria en la que cada pixel está representado por un cero (si es un pixel no sensitivo) o un uno (si el pixel es sensitivo).*
7. *En función de la matriz binaria recientemente generada, marcar en la imagen original las zonas sensitivas.*

{fin}

7.2.2 Soluciones paralelas utilizando pasaje de mensajes

A continuación se describen las soluciones implementadas que utilizan como modelo de programación paralela el pasaje de mensajes.

7.2.2.1 Solución tipo *pipelining* usando 5 núcleos

La primera solución tipo *pipelining* utiliza 5 procesos. Es una solución que utiliza como modelo de algoritmo paralelo paralelismo funcional puro.

{Para cada una de las imágenes}

Proceso 0:

1. *Separar la imagen en los tres canales de color (rojo, verde y azul).*
2. *Para cada canal de color comunicar los datos al proceso correspondiente.*

***Procesos 1, 2 y 3** ejecutan los mismos pasos cada uno para el canal de color que le corresponde:*

1. *Generar una imagen para su canal.*
2. *Aplicar un filtro gaussiano de 5X5 en tres niveles, generando una imagen para cada nivel.*

3. *Sumar las tres imágenes difuminadas, generando una nueva imagen.*
4. *Luego comunicar estos datos al proceso 4.*

Proceso 4:

1. *Unir las tres imágenes difuminadas generando una nueva imagen.*
2. *Convertir el formato de representación de la imagen recién generada de RGB a HSV.*
3. *Generar el umbral.*
4. *Generar la matriz binaria.*
5. *En función de la matriz binaria marcar en la imagen original las zonas sensitivas.*

{fin}

En la Fig. 7.1 puede verse un esquema del algoritmo.

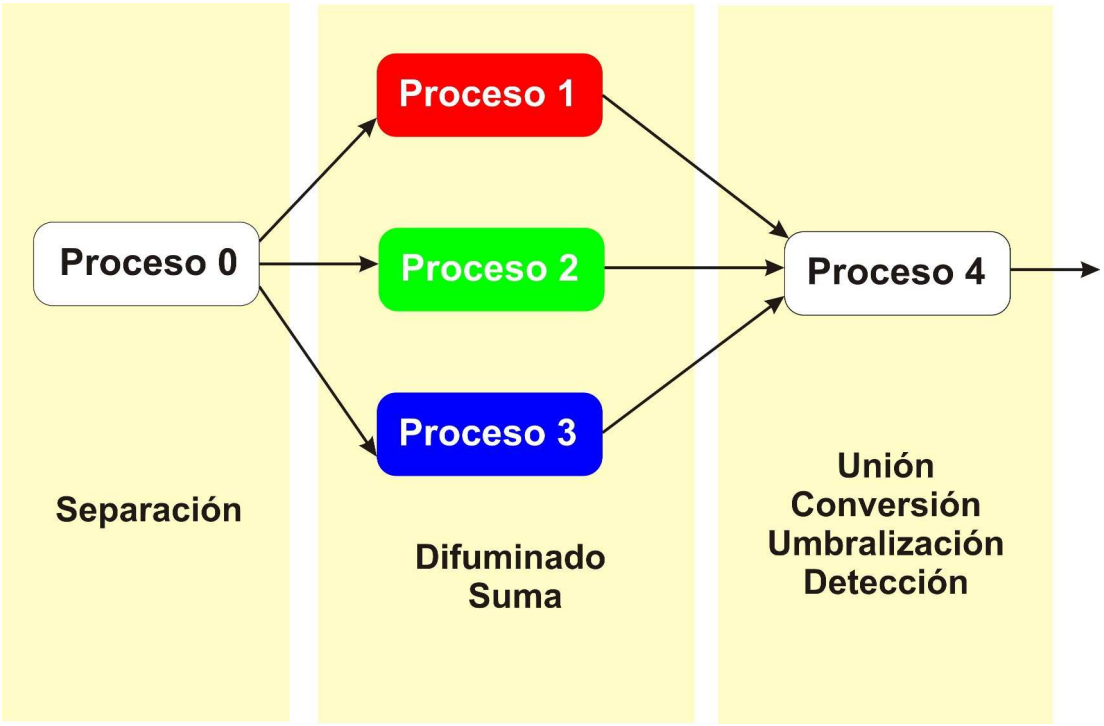

Fig. 7.1

7.2.2.2 Solución tipo *pipelining* usando 8 núcleos

Es una solución tipo *pipelining* implementada con 8 procesos. Utiliza como modelo de algoritmo paralelo el paralelismo funcional puro.

{Para cada una de las imágenes}

Proceso 0:

1. *Separar la imagen en los tres canales de color (rojo, verde y azul).*
2. *Para cada canal de color comunicar los datos al proceso correspondiente.*

Procesos 1, 2 y 3 *ejecutan los mismos pasos cada uno para el canal de color que le corresponde:*

1. *Generar una imagen para su canal.*
2. *Aplicar un filtro gaussiano de 5X5 en tres niveles, generando una imagen para cada nivel.*
3. *Sumar las tres imágenes difuminadas, generando una nueva imagen.*
4. *Luego comunicar estos datos al proceso 4.*

Proceso 4:

1. *Unir las tres imágenes difuminadas generando una nueva imagen.*
2. *Comunicar los datos de cada canal al proceso 5.*

Proceso 5:

1. *Convertir el formato de representación de la imagen recién generada de RGB a HSV.*
2. *Comunicar la matriz de brillo al proceso 6.*

Proceso 6:

1. *Generar el umbral.*
2. *Generar la matriz binaria.*
3. *Comunicar la matriz binaria al proceso 7.*

Proceso 7:

1. *En función de la matriz binaria marcar en la imagen original las zonas sensitivas.*

{fin}

A continuación, en la Fig. 7.2 puede verse un esquema del algoritmo.

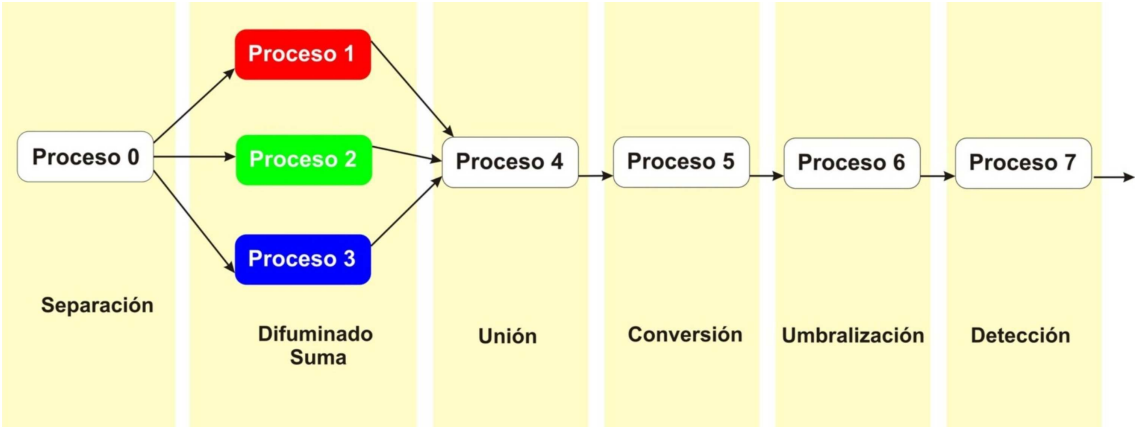

Fig. 7.2

7.2.2.3 Solución *Master-Worker* Replicada

Esta solución está implementada utilizando el paradigma de interacción entre procesos *Master – Worker*. Lleva a cabo una mezcla de paralelismo funcional y paralelismo de datos (replicación), replicando una vez el siguiente algoritmo:

{Para cada una de las imágenes}

Proceso 0:

1. *Separar la imagen en los tres canales de color (rojo, verde y azul).*
2. *Para cada canal de color comunicar los datos al proceso correspondiente.*
3. *Recibir los datos correspondientes a la unión de las imágenes difuminadas para cada canal de color.*
4. *Unir las tres imágenes difuminadas generando una nueva imagen.*
5. *Convertir el formato de representación de la imagen recién generada de RGB a HSV.*
6. *Generar el umbral.*
7. *Generar la matriz binaria.*
8. *En función de la matriz binaria marcar en la imagen original las zonas sensitivas.*

Procesos 1, 2 y 3 *ejecutan los mismos pasos cada uno para el canal de color que le corresponde:*

1. *Generar una imagen para su canal.*
2. *Aplicar un filtro gaussiano de 5X5 en tres niveles, generando una imagen para cada nivel.*
3. *Sumar las tres imágenes difuminadas, generando una nueva imagen.*
4. *Luego comunicar estos datos al proceso 0.*

{fin}

En la Fig. 7.3 se muestra un esquema del algoritmo.

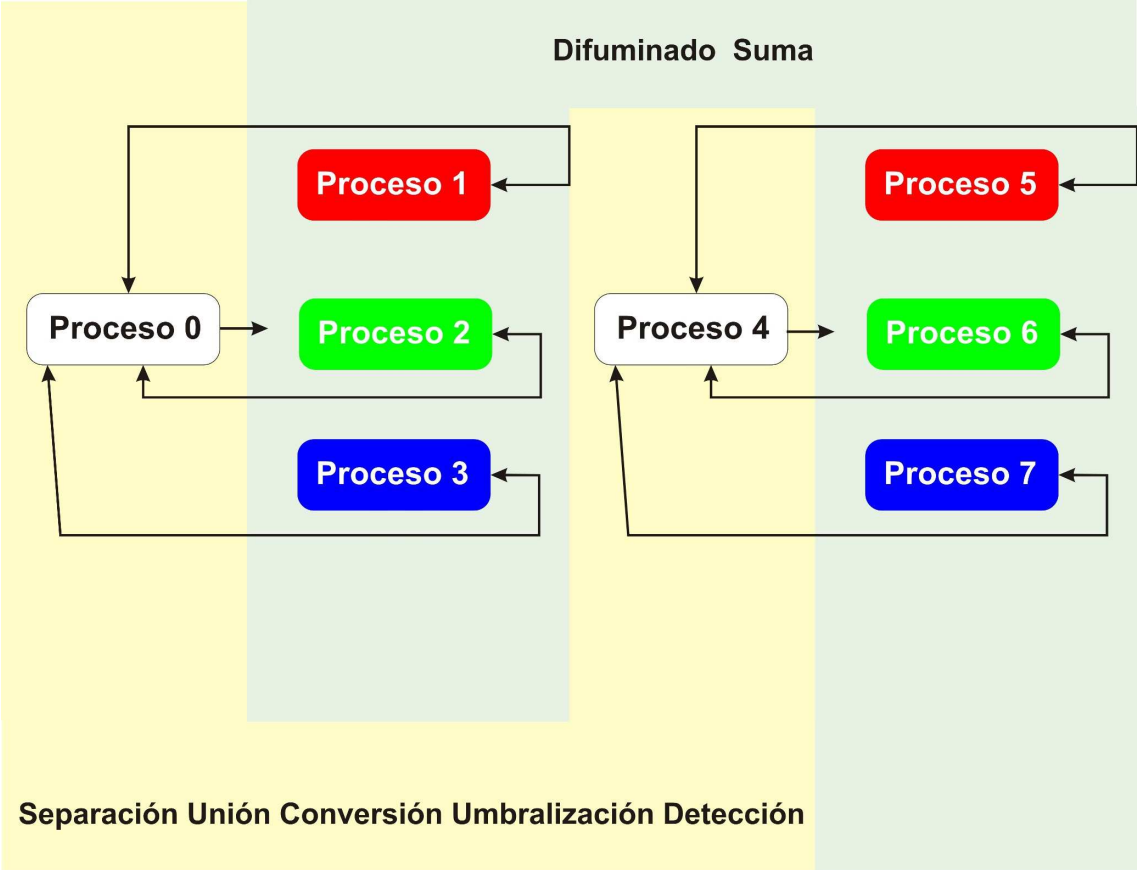

Fig. 7.3

7.2.2.4 Filtro gaussiano subdividido

Debido a que el procesamiento del filtro gaussiano es la etapa más larga dentro del algoritmo, en esta solución se desacopla el mismo para que su procesamiento pueda ser subdividido y de esta manera generar un *pipe* con

sus fases. En las soluciones anteriores, se utiliza un único proceso para llevar a cabo las fases de procesamiento propias de cada color. En esta, se desacopla el procesamiento en dos procesos, de manera de poder subdividir el procesamiento del filtro gaussiano y poder generar mayor nivel de paralelismo en el *pipe*.

Es una solución tipo *pipelining* implementada con 8 procesos. Utiliza como modelo de algoritmo paralelo el paralelismo funcional puro.

{Para cada una de las imágenes}

Proceso 0:

1. *Separar la imagen en los tres canales de color (rojo, verde y azul).*
2. *Generar imagen para cada canal.*
3. *Para cada canal de color comunicar los datos al proceso correspondiente.*

Procesos 1, 3 y 5:

1. *Aplicar dos fases de filtros gaussianos.*
2. *Comunicar los datos al proceso correspondiente.*

Procesos 2, 4, y 6:

1. *Aplica el tercer filtro gaussiano.*
2. *Sumar las tres imágenes difuminadas, generando una nueva imagen.*
3. *Comunicar los datos al Proceso 7.*

Proceso 7:

1. *Unir las tres imágenes difuminadas generando una nueva imagen.*
2. *Convertir el formato de representación de la imagen recién generada de RGB a HSV.*
3. *Generar el umbral.*
4. *Generar la matriz binaria.*

5. *En función de la matriz binaria marcar en la imagen original las zonas sensitivas.*

{fin}

En la Fig. 7.4 puede verse un esquema del algoritmo.

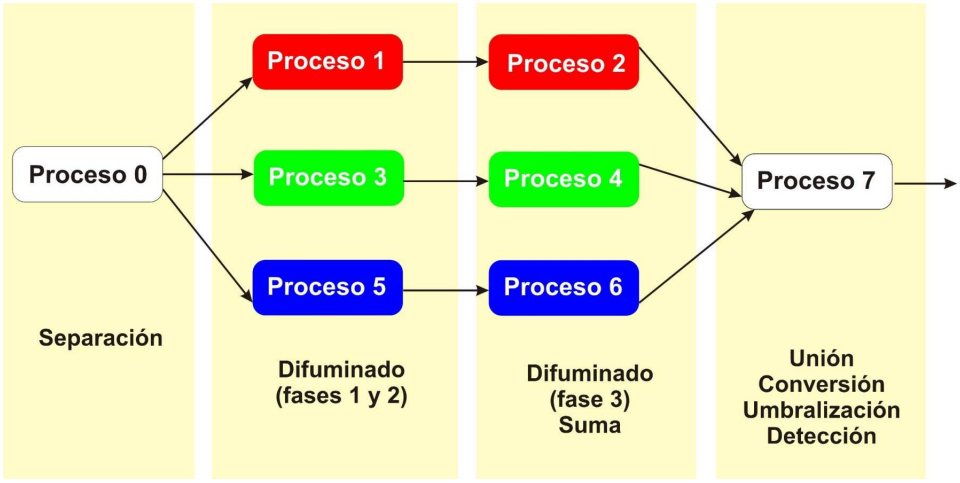

Fig. 7.4

7.2.2.5 *BASIZ* - ParDatos *MPI*

La solución que a continuación se describe utiliza como estrategia de descomposición el paralelismo de datos.

Si se toma a la imagen como una matriz, el algoritmo divide la misma en filas para ser procesadas. Cada proceso se encarga de llevar a cabo las fases de procesamiento sobre las filas que le fueron asignadas, y luego le envía a un proceso coordinador los datos procesados, para que se ejecuten las últimas fases del algoritmo que requiere de todas las filas.

{Para cada una de las imágenes}

Proceso 0:

1. *Separar la imagen en los tres canales de color (rojo, verde y azul).*

Proceso 0 – (N-1):

1. *Generar una imagen para cada canal de color.*

2. *Aplicar el difuminado a los tres canales de color. Para cada canal se aplica un filtro gaussiano de 5X5 en tres niveles, generando una imagen para cada nivel.*

3. *Para cada canal de color, sumar las tres imágenes difuminadas generando una nueva imagen.*

4. *Unir las tres imágenes difuminadas generando una nueva imagen.*

5. *Convertir el formato de representación de la imagen recién generada de RGB a HSV (matiz, saturación y brillo).*

6. *Enviar datos al proceso coordinador*

Proceso N-1:

1. *Generar el umbral a partir del cual se determinará si un pixel pertenece o no a una zona sensitiva.*

2. *Generar una matriz binaria en la que cada pixel está representado por un cero (si es un pixel no sensitivo) o un uno (si el pixel es sensitivo).*

3. *En función de la matriz binaria recientemente generada, marcar en la imagen original las zonas sensitivas.*

{fin}

En la Fig. 7.5 puede verse un esquema del algoritmo.

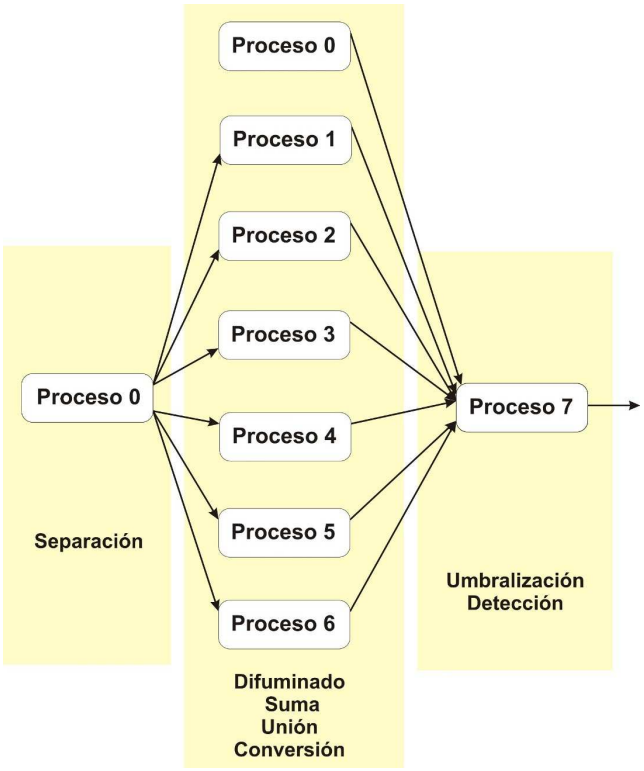

Fig. 7.5

7.2.3 Soluciones paralelas utilizando memoria compartida

A continuación se describen las soluciones implementadas que utilizan como modelo de programación paralela memoria compartida.

7.2.3.1 BASIZ – ParDatos OpenMP

Esta solución utiliza como estrategia de descomposición el paralelismo de datos puro.

El algoritmo divide la imagen en filas para ser procesadas. Cada hilo se encarga de llevar a cabo las fases de procesamiento sobre las filas que le fueron asignadas, y luego le envía a un hilo coordinador los datos procesados, para que se ejecuten las últimas fases del algoritmo que requiere de todas las filas.

{Para cada una de las imágenes}

Main:

1. *Separar la imagen en los tres canales de color (rojo, verde y azul).*

Hilos 0 – (N-1):

1. *Generar una imagen para cada uno de ellos.*
2. *Aplicar el difuminado a los tres canales de color. Para cada canal se aplica un filtro gaussiano de 5X5 en tres niveles, generando una imagen para cada nivel.*
3. *Para cada canal de color, sumar las tres imágenes difuminadas generando una nueva imagen.*
4. *Unir las tres imágenes difuminadas generando una nueva imagen.*
5. *Convertir el formato de representación de la imagen recién generada de RGB a HSV (matiz, saturación y brillo).*

2. *Generar el umbral a partir del cual se determinará si un pixel pertenece o no a una zona sensitiva.*
3. *Generar una matriz binaria en la que cada pixel está representado por un cero (si es un pixel no sensitivo) o un uno (si el pixel es sensitivo).*
4. *En función de la matriz binaria recientemente generada, marcar en la imagen original las zonas sensitivas.*

{fin}

En la Fig. 7.6 puede verse un esquema del algoritmo.

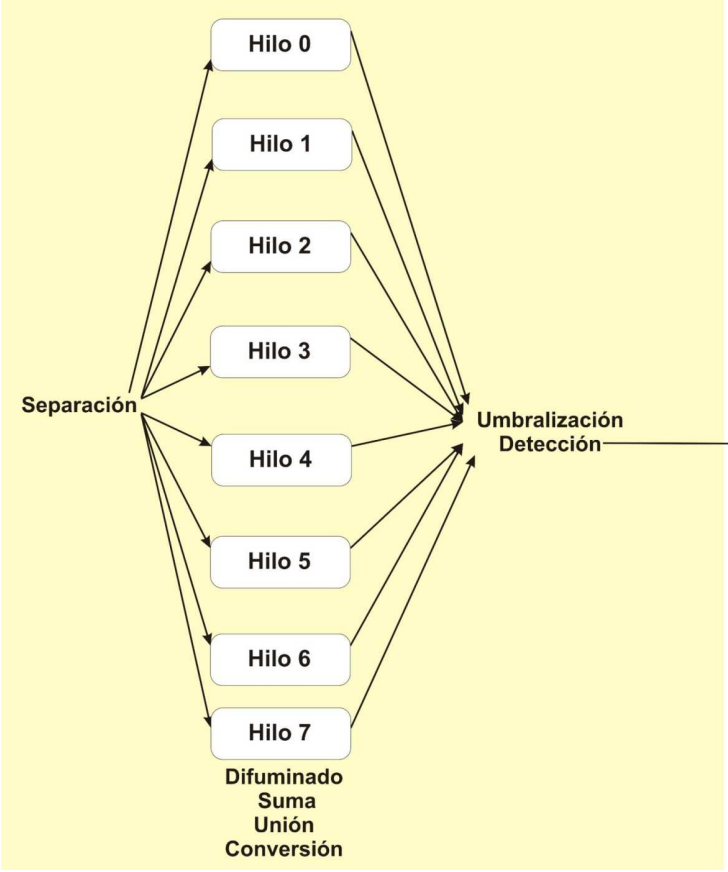

Fig. 7.6

7.2.3.2 BASIZ – ParDatos OpenMP dos fases

Esta solución es similar a la descripta en la Sección 7.2.3.1 con la diferencia de que luego de calculado el umbral, el resto de las fases se paralelizan de la misma manera que en las fases previas. Se utiliza también el paralelismo de datos puro.

{Para cada una de las imágenes}

Main:

1. *Separar la imagen en los tres canales de color (rojo, verde y azul).*

Hilos 0 – (N-1):

1. *Generar una imagen para cada canal.*

2. *Aplicar el difuminado a los tres canales de color. Para cada canal se aplica un filtro gaussiano de 5X5 en tres niveles, generando una imagen para cada nivel.*

3. *Para cada canal de color, sumar las tres imágenes difuminadas generando una nueva imagen.*

4. *Unir las tres imágenes difuminadas generando una nueva imagen.*

5. *Convertir el formato de representación de la imagen recién generada de RGB a HSV (matiz, saturación y brillo).*

2. *Generar el umbral a partir del cual se determinará si un pixel pertenece o no a una zona sensitiva.*

Hilos 0 – (N-1):

1. *Generar una matriz binaria en la que cada pixel está representado por un cero (si es un pixel no sensitivo) o un uno (si el pixel es sensitivo).*

2. *En función de la matriz binaria recientemente generada, marcar en la imagen original las zonas sensitivas.*

{fin}

En la Fig. 7.7 puede verse un esquema del algoritmo.

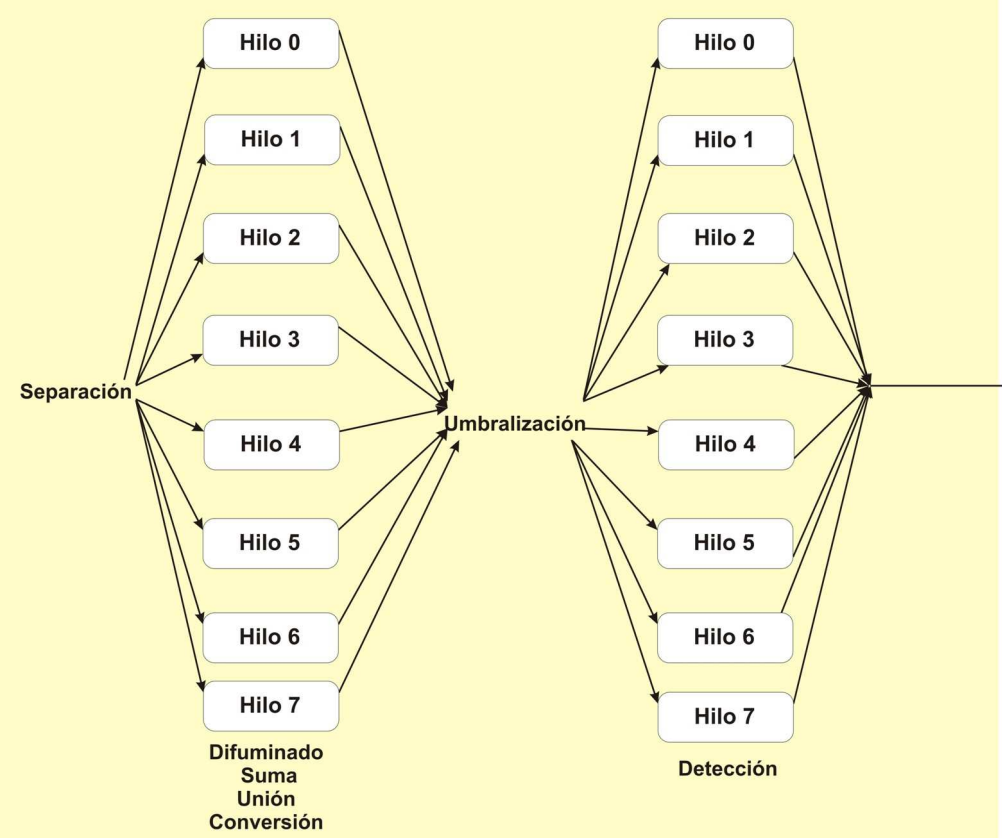

Fig. 7.7

7.2.4 Soluciones paralelas utilizando memoria compartida y pasaje de mensajes: programación híbrida

A continuación se describen las soluciones implementadas que utilizan como modelo de programación paralela memoria compartida y pasaje de mensajes.

7.2.4.1 Basiz-Pipe-5 MPI y Pthreads

Es una solución tipo *pipelining* implementada con 2 procesos que se comunican mediante pasaje de mensajes. Dentro del proceso 0 se ejecutan 3 hilos (*Pthreads*), uno para cada color. Utiliza como modelo de algoritmo paralelo el paralelismo funcional puro.

{Para cada una de las imágenes}

Proceso 0:

1. *Separar la imagen en los tres canales de color (rojo, verde y azul).*

Hilos 1, 2 y 3 *ejecutan los mismos pasos cada uno para el canal de color que le corresponde:*

1. *Generar una imagen para su canal.*
2. *Aplicar un filtro gaussiano de 5X5 en tres niveles, generando una imagen para cada nivel.*
3. *Sumar las tres imágenes difuminadas, generando una nueva imagen.*

2. *Comunicar los datos correspondientes a cada canal de color al Proceso 1.*

Proceso 1:

1. *Unir las tres imágenes difuminadas generando una nueva imagen.*
2. *Convertir el formato de representación de la imagen recién generada de RGB a HSV.*
3. *Generar el umbral.*
4. *Generar la matriz binaria.*
5. *En función de la matriz binaria marcar en la imagen original las zonas sensitivas.*

{fin}

En la Fig. 7.8 puede verse un esquema del algoritmo.

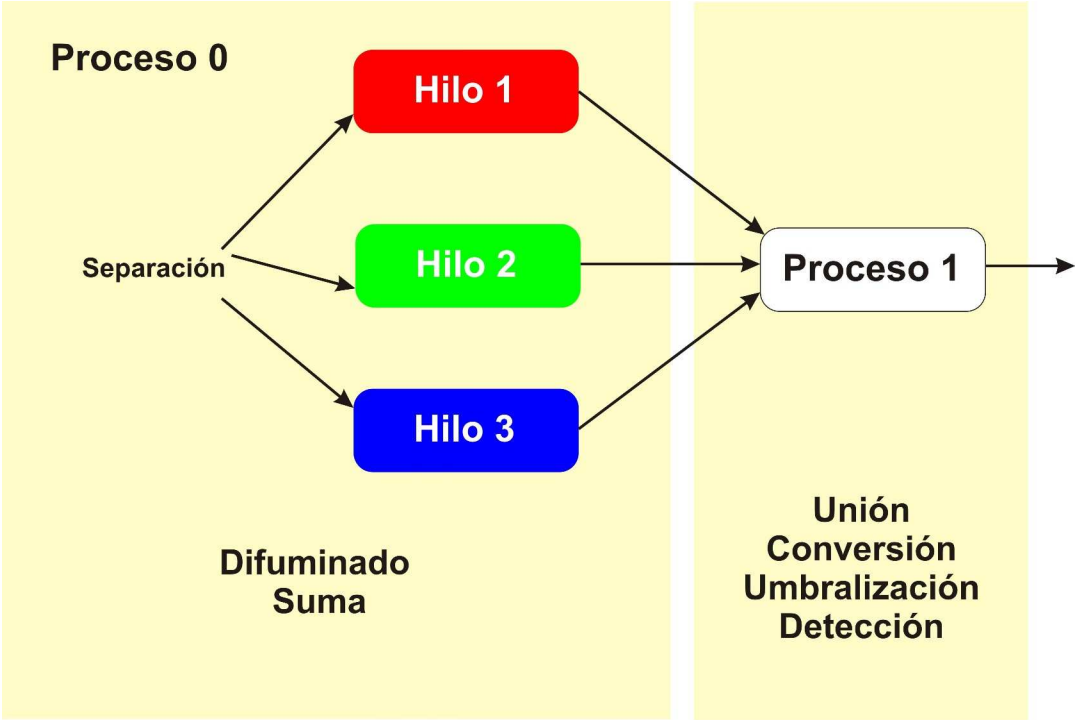

Fig. 7.8

7.2.4.2 Basiz-Pipe-5 MPI y OpenMP

Es una solución tipo *pipelining* implementada con 2 procesos que se comunican mediante pasaje de mensajes. Dentro del proceso 0 se ejecutan 3 hilos (*OpenMP*), uno para cada color. Utiliza como modelo de algoritmo paralelo el paralelismo funcional puro.

{Para cada una de las imágenes}

Proceso 0:

1. *Separar la imagen en los tres canales de color (rojo, verde y azul).*

Hilos 1, 2 y 3 *ejecutan los mismos pasos cada uno para el canal de color que le corresponde:*

1. *Generar una imagen para su canal.*
2. *Aplicar un filtro gaussiano de 5X5 en tres niveles, generando una imagen para cada nivel.*
3. *Sumar las tres imágenes difuminadas, generando una nueva imagen.*

2. *Comunicar los datos correspondientes a cada canal de color al Proceso 1.*

Proceso 1:

1. *Unir las tres imágenes difuminadas generando una nueva imagen.*
2. *Convertir el formato de representación de la imagen recién generada de RGB a HSV.*
3. *Generar el umbral.*
4. *Generar la matriz binaria.*
5. *En función de la matriz binaria marcar en la imagen original las zonas sensitivas.*

{fin}

En la Fig. 7.8 pudo verse un esquema del algoritmo.

8 Resultados obtenidos

A continuación, se presentarán los tiempos de ejecución obtenidos para cada una de las soluciones explicadas en la Sección 7.2. En todos los casos, las pruebas fueron realizadas utilizando imágenes de 4096 X 4096 pixeles.

El hardware que se utiliza para llevar a cabo las pruebas es un *Multicore Dell Poweredge* 1950, que posee 2 procesadores *quad core Intel Ceón* e5410 de 2.33 *GHz*; 4 *Gb* de memoria *RAM* (compartida entre ambos procesadores); *cache* L2 de *6Mb* entre cada par de núcleos de los procesadores. El sistema operativo que se utiliza es *Fedora* 11 de 32 *bits*.

Los tiempos de ejecución fueron tomados para diferente cantidad de imágenes procesadas y están dados en segundos.

8.1 Solución secuencial

Nº de imágenes	Tiempo de ejecución
16	1096,184349
18	1236,224733
20	1381,812570
22	1527,840057
24	1664,409866
26	1809,448193
28	1940,566717
30	2072,817969

8.2 Soluciones paralelas utilizando pasaje de mensajes

8.2.1 Solución tipo *pipelining* usando 5 núcleos

Nº de imágenes	*BASIZ-Pipe-5 Mapping* SO	*BASIZ-Pipe-5 Mapping* manual
16	347,115726	352,381197
18	396,351962	**391,463422**
20	432,877443	434,128574
22	479,019390	486,018654
24	520,146222	**518,409603**
26	561,853795	562,451366
28	602,108146	606,043863
30	645,601332	647,988962

En la Fig. 8.1 puede verse un esquema del *mapping* manual realizado.

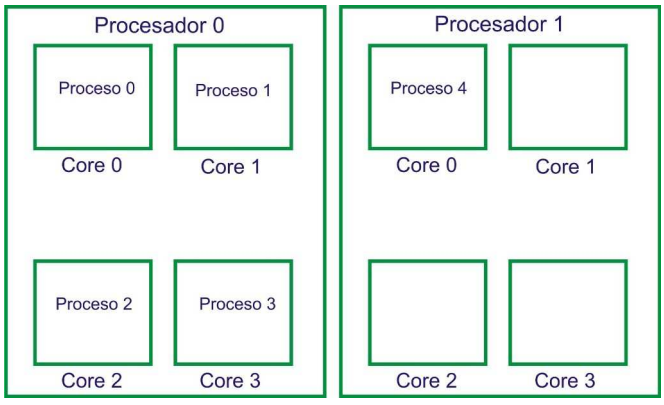

Fig. 8.1

8.2.2 Solución tipo *pipelining* usando 8 núcleos

Nº de imágenes	*BASIZ-Pipe-8 Mapping* SO	*BASIZ-Pipe-8 Mapping* manual
16	350,404372	350,953746
18	390,577530	392,367580
20	433,501823	435,226090
22	475,780649	484,694018
24	524,983332	**521,461347**
26	564,203506	**563,111458**
28	610,901599	**608,587703**
30	647,120775	649,378497

En la Fig. 8.2 puede verse un esquema del *mapping* manual realizado.

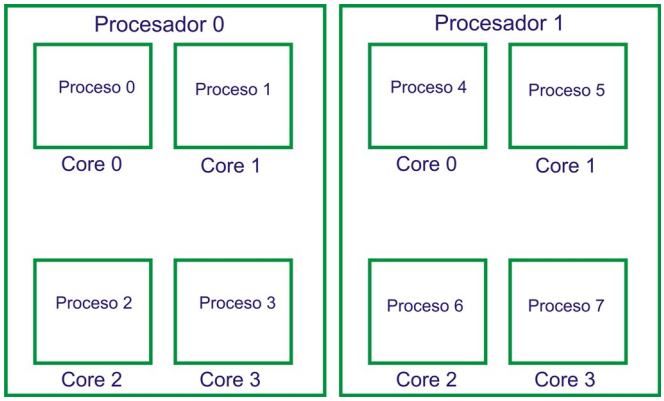

Fig. 8.2

8.2.3 Solución *Master-Worker* Replicada

Nº de imágenes	*BASIZ - MW* – Replicado *Mapping* SO	*BASIZ - MW* – Replicado *Mapping* manual
16	252,665408	**249,079594**
18	281,888035	**280,753140**
20	314,353628	316,070032
22	350,136947	**342,358989**
24	378,151115	**376,240835**
26	414,803686	**409,411892**
28	437,716598	437,219655
30	473,109013	**462,835451**

En la Fig. 8.3 puede verse un esquema del *mapping* manual realizado.

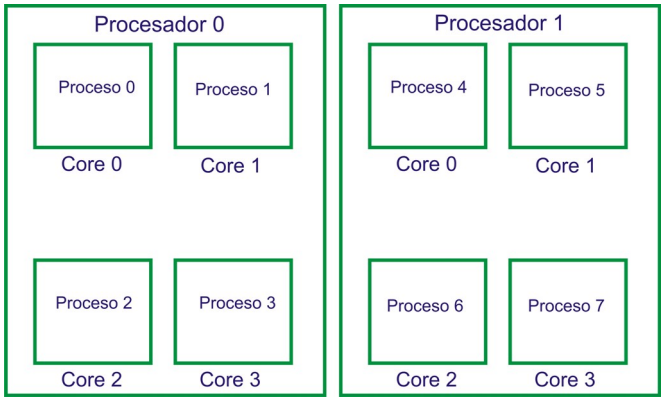

Fig. 8.3

8.2.4 Filtro gaussiano subdividido

Nº de imágenes	*BASIZ* - Gauss-subdividido *Mapping* SO	*BASIZ* - Gauss-subdividido *Mapping* manual
16	223,153278	**222,344147**
18	252,125920	**249,068975**
20	275,514801	**273,073403**
22	301,619722	**300,104738**
24	334,042334	**326,925126**
26	356,476677	**354,543364**
28	382,028761	**380,568260**
30	408,266502	**406,846495**

En la Fig. 8.4 puede verse un esquema del *mapping* manual realizado.

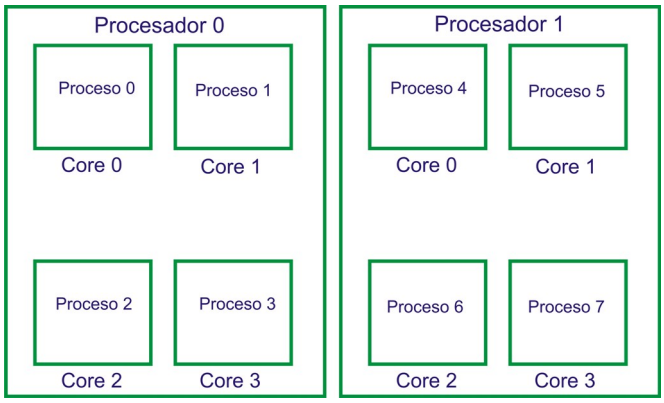

Fig. 8.4

8.2.5 *BASIZ* - ParDatos *MPI*

Nº de imágenes	*BASIZ* - Paralelismo Datos *MPI* *Mapping* SO	*BASIZ* - Paralelismo Datos *MPI* *Mapping* manual
16	212,936344	215,445463
18	242,696445	241,430224
20	267,001724	263,524395
22	298,280783	294,413336
24	322,785236	323,595153
26	344,577258	346,488119
28	374,040192	376,949338
30	404,676615	404,255814

En la Fig. 8.5 puede verse un esquema del *mapping* manual realizado.

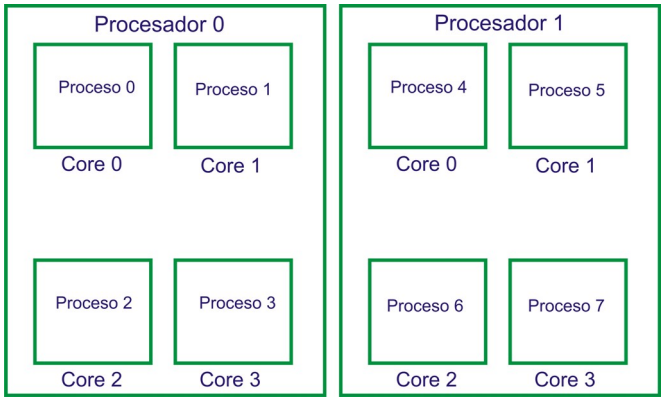

Fig. 8.5

8.3 Soluciones paralelas utilizando memoria compartida

8.3.1 *BASIZ* – ParDatos *OpenMP*

Nº de imágenes	*BASIZ* - Paralelismo Datos *OpenMP* *Mapping* SO	*BASIZ* - Paralelismo Datos *OpenMP* *Mapping* manual
16	334,849048	**321,166661**
18	373,997115	**370,915791**
20	410,507344	**401,345172**
22	456,516699	**440,880090**
24	496,848214	**488,258487**
26	536,998671	**519,898315**
28	578,313862	**575,305528**
30	619,769239	**605,592327**

En la Fig. 8.6 puede verse un esquema del *mapping* manual realizado.

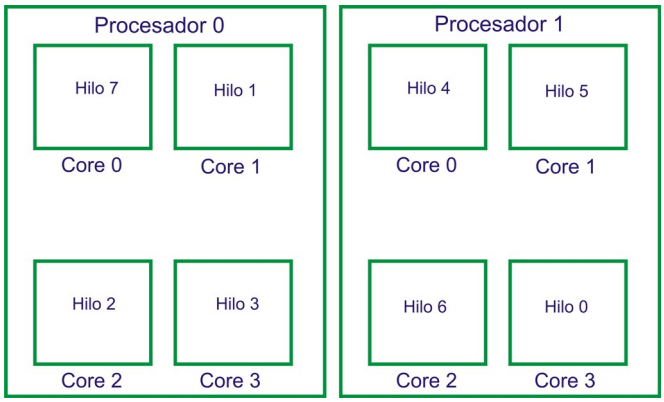

Fig. 8.6

8.3.2 *BASIZ* – ParDatos *OpenMP* dos fases

Nº de imágenes	*BASIZ* - Paralelismo Datos *OpenMP* 2 fases *Mapping* SO	*BASIZ* - Paralelismo Datos *OpenMP* 2 fases *Mapping* manual
16	327,020137	**319,208008**
18	371,435091	**361,295441**
20	406,152330	**402,738264**
22	451,149429	**427,888811**
24	491,852149	**481,836838**
26	526,920649	**522,117247**
28	570,474187	**554,244383**
30	611,421787	**597,155986**

En la Fig. 8.7 puede verse un esquema del *mapping* manual realizado.

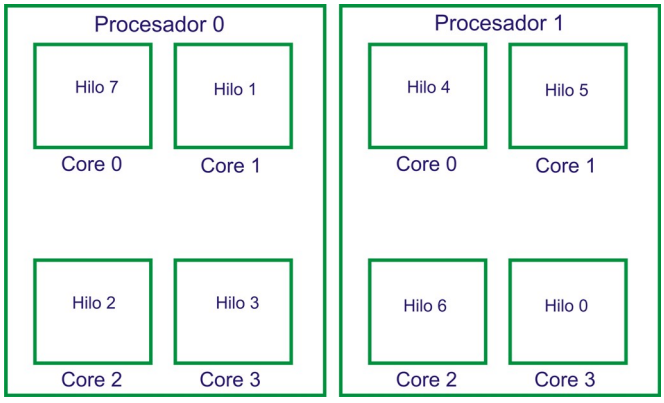

Fig. 8.7

8.4 Soluciones paralelas utilizando memoria compartida y pasaje de mensajes: programación híbrida

8.4.1 BASIZ-Pipe-5 MPI y Pthreads

Nº de imágenes	*BASIZ-Pipe-5 MPI* y *Pthreads* *Mapping* SO	*BASIZ-Pipe-5 MPI* y *Pthreads* *Mapping* manual
16	642,582053	644,249704
18	718,571373	721,281251
20	802,671450	**801,384805**
22	880,053590	**879,821482**
24	962,897406	**959,197701**
26	1043,167424	**1042,468147**
28	1124,005201	**1121,822797**
30	1203,198628	**1198,682953**

En la Fig. 8.8 puede verse un esquema del *mapping* manual realizado.

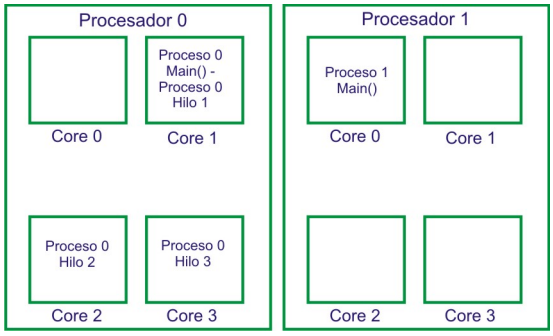

Fig. 8.8

8.4.2 *BASIZ*-Pipe-5 *MPI* y *OpenMP*

Nº de imágenes	*BASIZ-Pipe-5 MPI* y *OpenMP* *Mapping* SO	*BASIZ-Pipe-5 MPI* y *OpenMP* *Mapping* manual
16	613,662922	**612,254646**
18	686,507809	687,363345
20	762,644351	763,092738
22	839,522619	**837,767049**
24	916,426840	**913,377305**
26	990,880144	990,648108
28	1067,840504	1067,194910
30	1144,318844	**1139,923248**

En la Fig. 8.9 puede verse un esquema del *mapping* manual realizado.

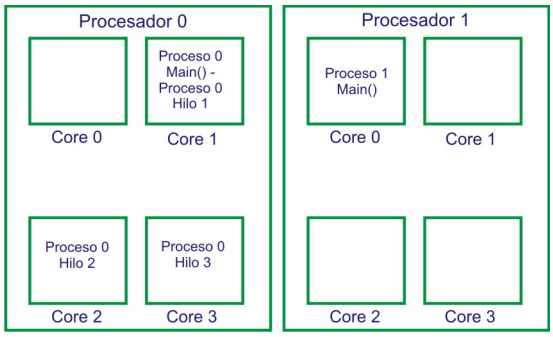

Fig. 8.9

8.5 Resultados comparados

A continuación se observan los gráficos del *speedup* y la eficiencia de los algoritmos recientemente explicados. En ellos se contrastan el *speedup* y la eficiencia obtenidos con *mapping* del sistema operativo y con *mapping* manual.

Además se muestran los gráficos que representan el porcentaje de diferencia entre la asignación automática y manual de las alternativas implementadas: ((Tiempo automático – Tiempo manual) *100/Tiempo automático).

8.5.1 Soluciones que utilizan paralelismo funcional (pasaje de mensajes)

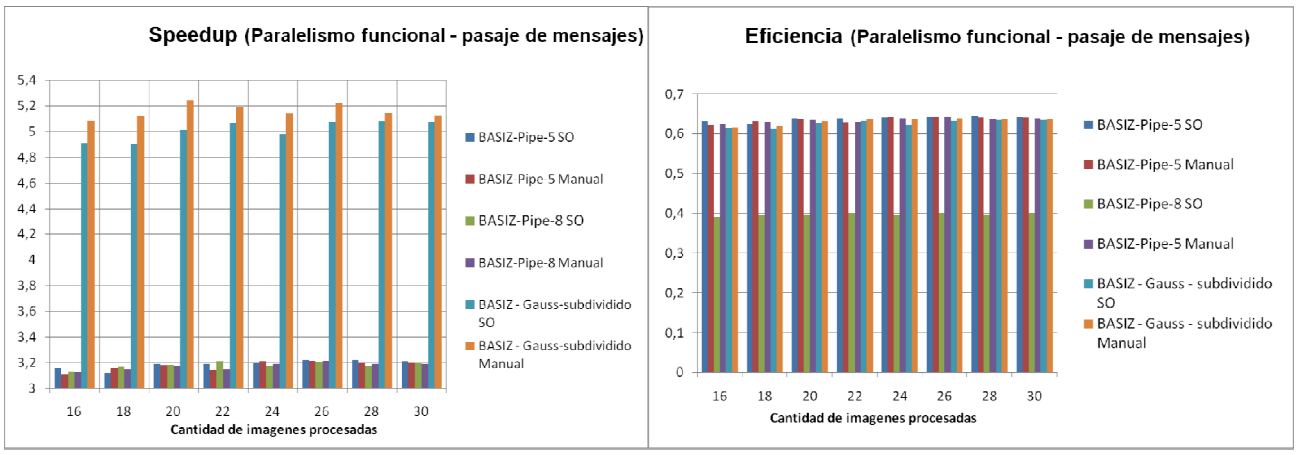

Fig. 8.10

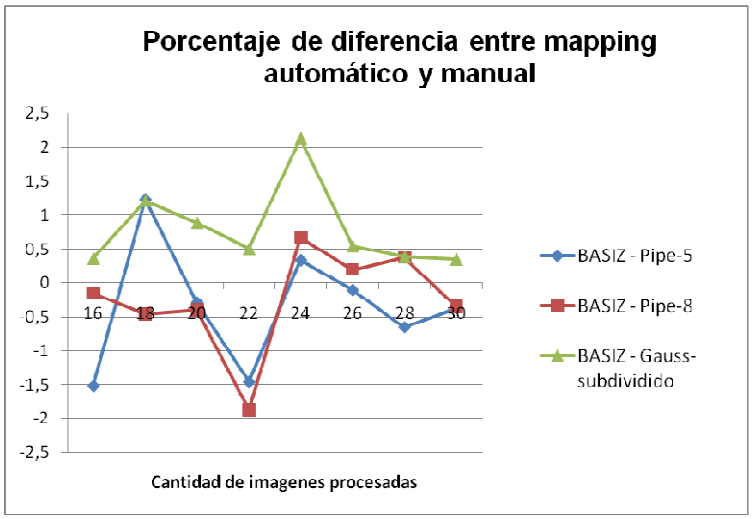

Fig. 8.11

8.5.2 Soluciones que utilizan paralelismo de datos

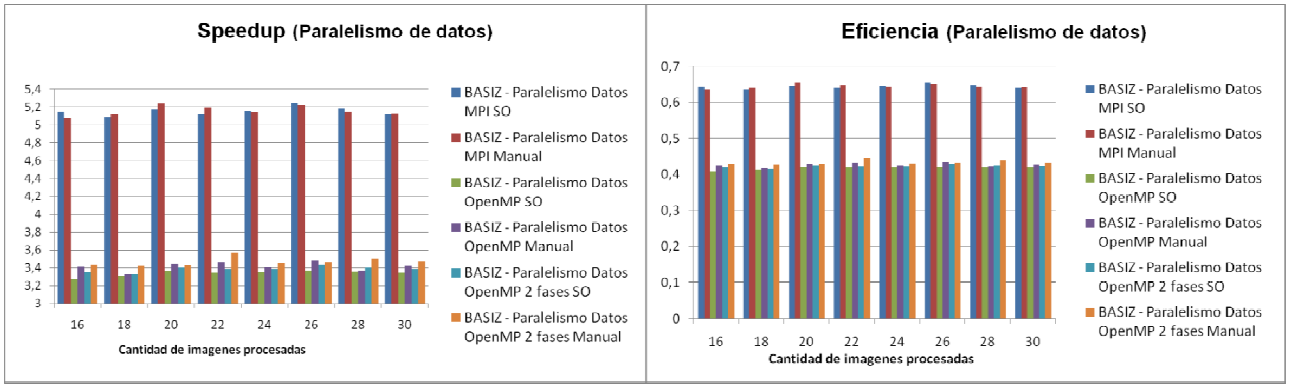

Fig. 8.12

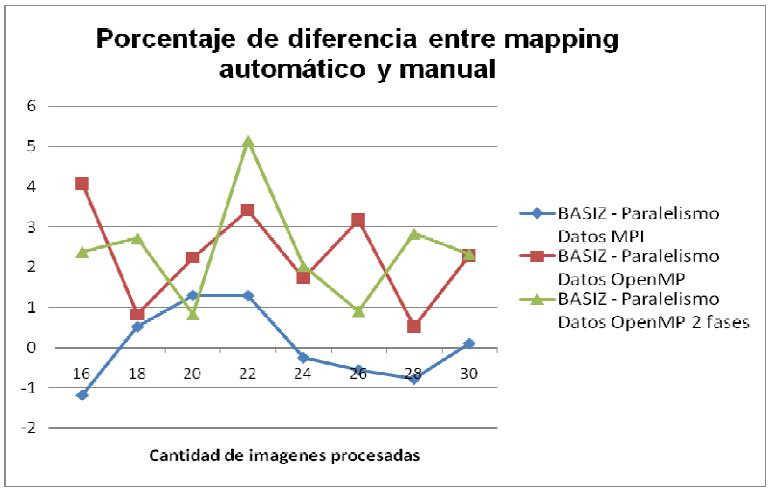

Fig. 8.13

8.5.3 Soluciones que utilizan paralelismo funcional y de datos

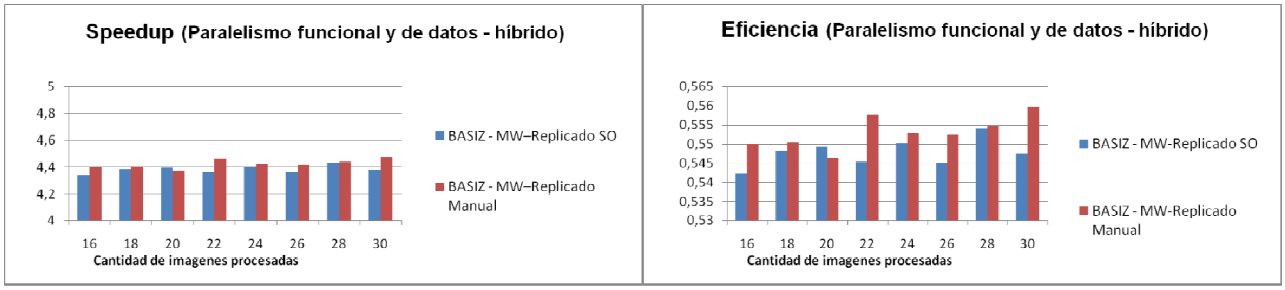

Fig. 8.14

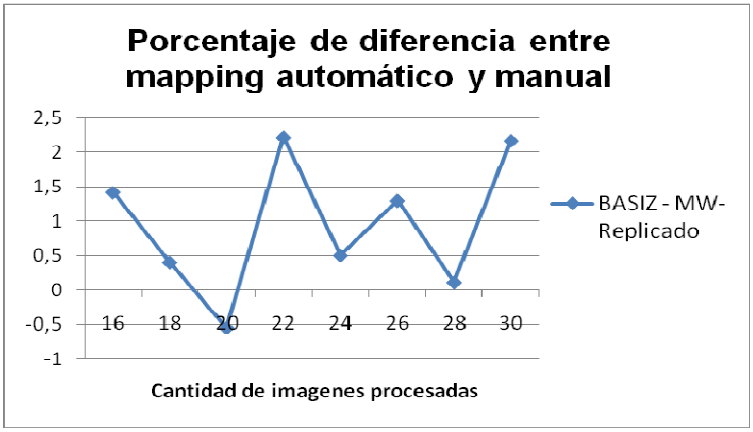

Fig. 8.15

8.5.4 Soluciones que utilizan paralelismo funcional (memoria compartida y pasaje de mensajes)

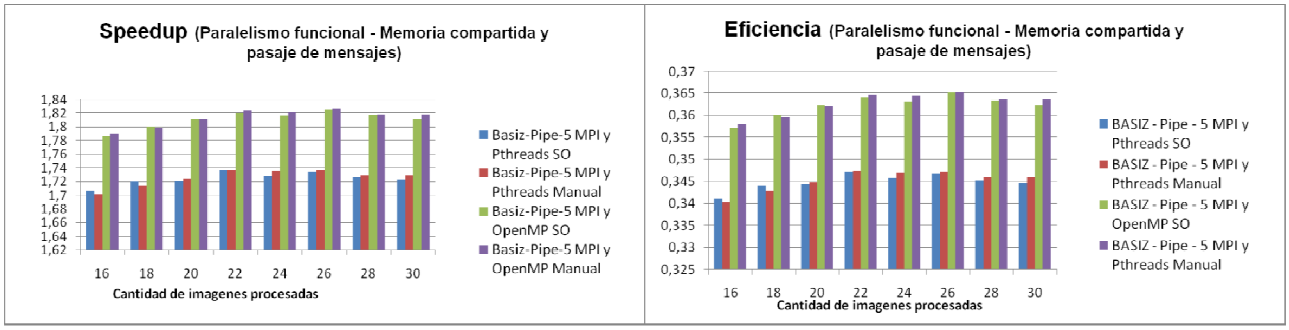

Fig. 8.16

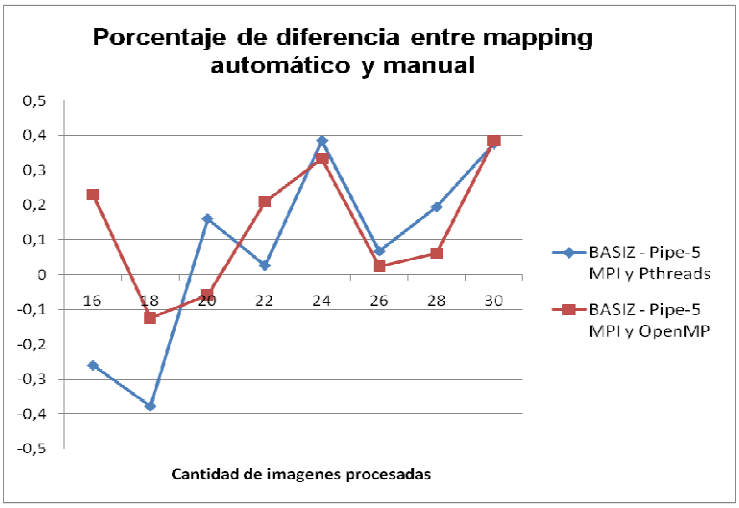

Fig. 8.17

9 Solución al problema en *cluster* de *multicore*

A continuación se describe una solución al problema *BASIZ* en la que se utilizan 64 núcleos de procesamiento. Esta solución está pensada para ser ejecutada en una arquitectura de *cluster* de *multicore.* Esta implementación y las pruebas respectivas realizadas contribuyen al análisis planteado dentro de los objetivos de la Tesina, y brindan nuevas líneas de investigación futura.

El hardware que se utiliza para llevar a cabo las pruebas es un *Blade* de 8 servidores (hojas). Cada hoja posee 2 procesadores *quad core Intel Xeón* e5405 de 2.0 *GHz*; 2 *Gb* de memoria *RAM* (compartida entre ambos procesadores); *cache* L2 de 2 X *6Mb* compartida entre cada par de *cores* por procesador. El sistema operativo que se utiliza es *Fedora* 12 de 64 *bits*.

9.1 Filtro gaussiano subdividido utilizando 64 núcleos

En la Sección 7.2.2.4 se describe una solución al problema *BASIZ* en la cual la fase de procesamiento del filtro gaussiano es dividida en dos procesos. En este caso, esta fase es dividida en tres procesos y además las fases de conversión, umbralización, y detección son procesadas por un proceso cada una. Se utiliza como modelo de algoritmo paralelo una mezcla de paralelismo funcional y paralelismo de datos (replicación), replicando tres veces el siguiente algoritmo.

{Para cada una de las imágenes}

Proceso 0:

1. *Separar la imagen en los tres canales de color (rojo, verde y azul).*
2. *Generar imagen para cada canal de color.*
3. *Para cada canal de color comunicar los datos al proceso correspondiente.*

Procesos 1, 2, 3, 17, 18, 19, 33, 34, 35, 49, 50 y 51:

1. *Aplicar el primer filtro gaussiano.*
2. *Comunicar los datos al proceso correspondiente.*

Procesos 4, 5, 6, 20, 21, 22, 36, 37, 38, 52, 53 y 54:

1. *Aplica el segundo filtro gaussiano.*
2. *Comunicar los datos al proceso correspondiente.*

Procesos 7, 8, 9, 23, 24, 25, 39, 40, 41, 55, 56, y 57:

1. *Aplica el tercer filtro gaussiano.*
2. *Comunicar los datos al proceso correspondiente.*

Procesos 10, 11, 12, 26, 27, 28, 42, 43, 44, 58, 59, 60:

1. *Sumar las tres imágenes difuminadas, generando una nueva imagen.*
2. *Comunicar los datos al proceso correspondiente.*

Procesos 13, 29, 45, 61:

1. *Unir las tres imágenes difuminadas generando una nueva imagen.*
2. *Convertir el formato de representación de la imagen recién generada de RGB a HSV.*

Procesos 14, 30, 46, 62:

1. *Generar el umbral.*
2. *Generar la matriz binaria.*

Procesos 15, 31, 47, 63:

1. *En función de la matriz binaria marcar en la imagen original las zonas sensitivas.*

{fin}

En la Fig. 9.1 puede verse un esquema del algoritmo.

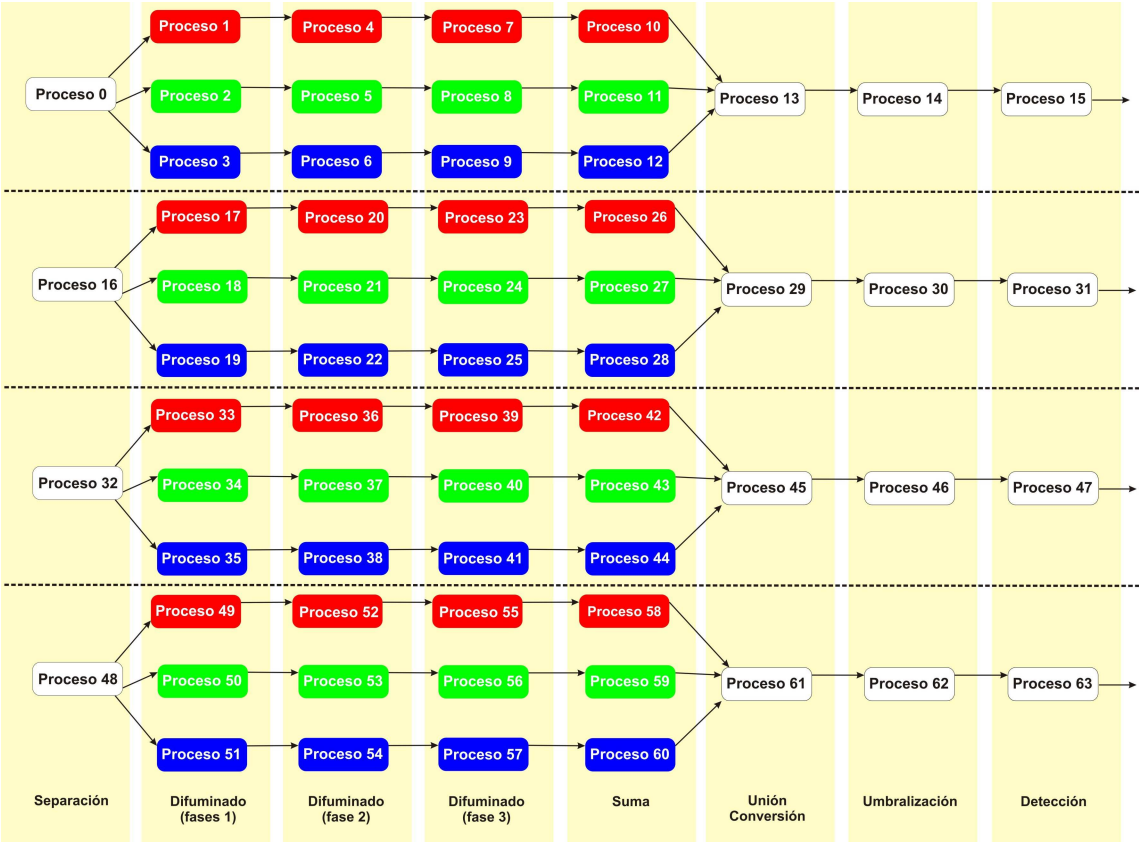

Fig. 9.1

9.2 Resultados obtenidos

Nº de imágenes	*BASIZ* - Gauss-subdividido *Mapping* SO	*BASIZ* - Gauss-subdividido *Mapping* manual	
		Alternativa 1	*Alternativa 2*
16	319,181253	**313,441675**	**313,667377**
20	399,123252	**385,129920**	**387,786698**
24	461,319151	469,306252	**458,643701**
28	554,415960	**553,611378**	**543,976638**
32	629,045266	**626,336235**	**622,197859**

En la Fig. 9.2 puede verse un esquema del *mapping* manual realizado (alternativa 1).

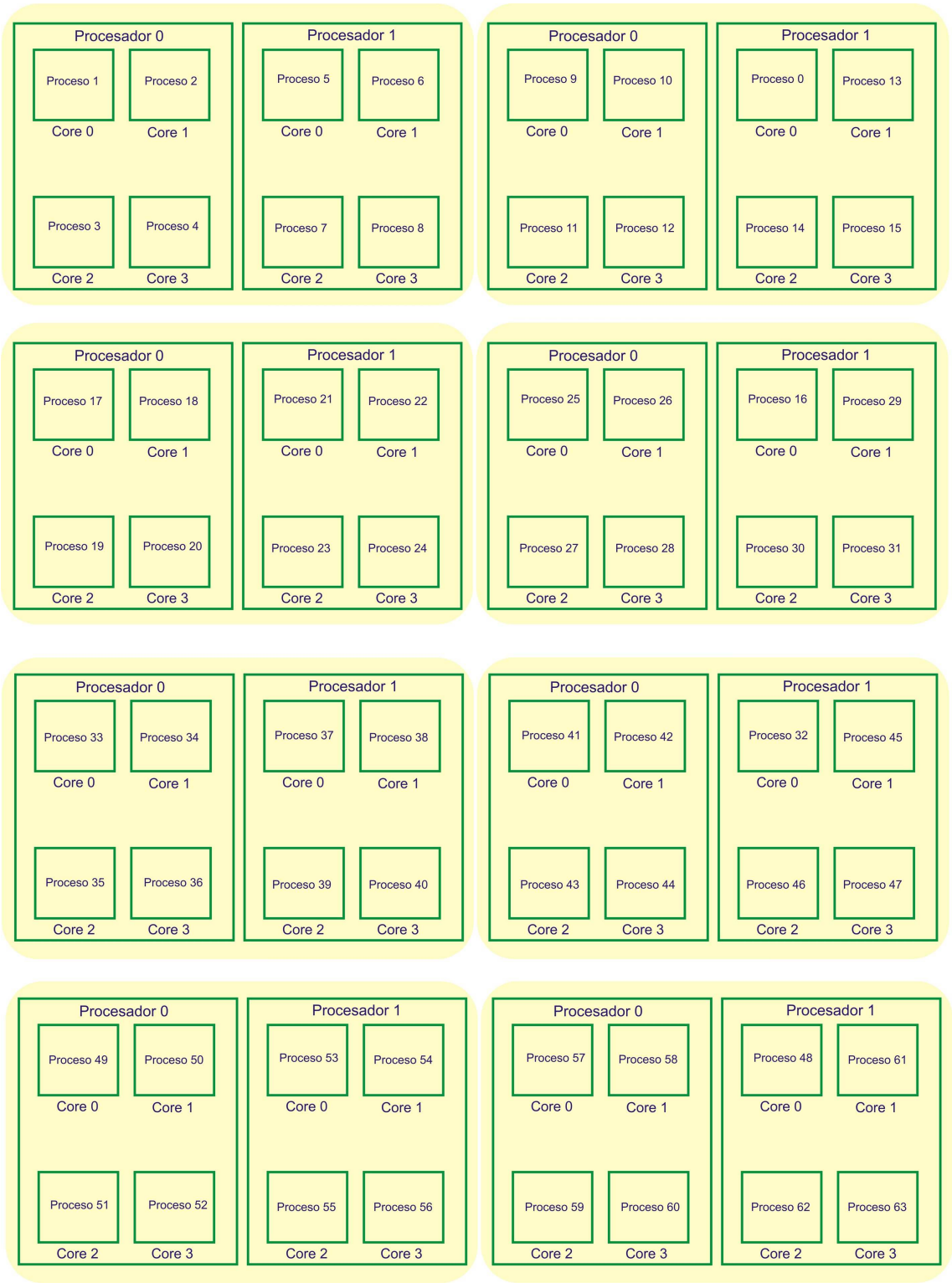

Fig. 9.2

En la Fig. 9.3 puede verse un esquema del *mapping* manual realizado (alternativa 2).

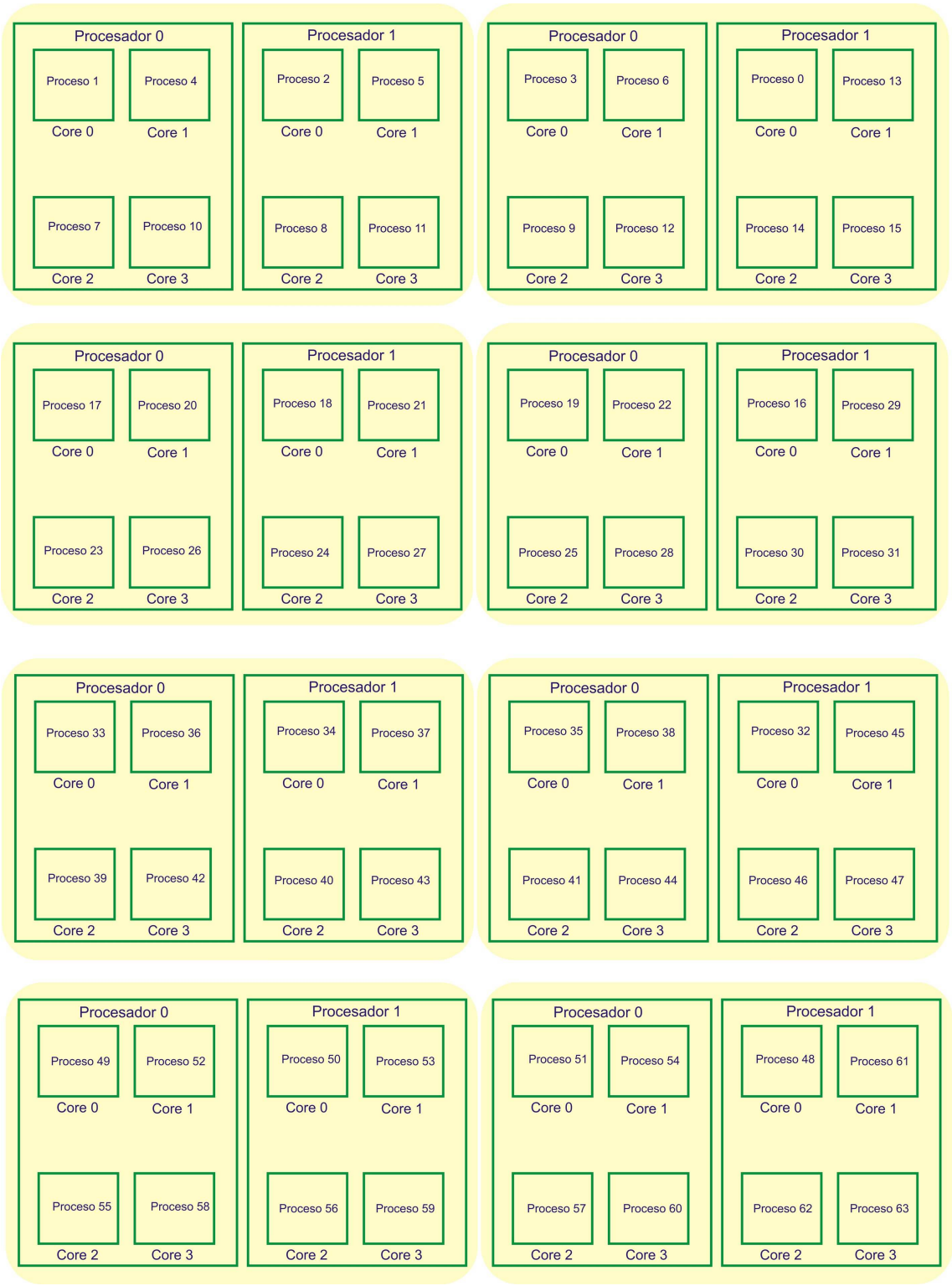
Procesador 0
Proceso 1
Proceso 4
Core 0
Core 1
Proceso 7
Proceso 10
Core 2
Core 3
Procesador 1
Proceso 2
Proceso 5
Core 0
Core 1
Proceso 8
Proceso 11
Core 2
Core 3
Procesador 0
Proceso 3
Proceso 6
Core 0
Core 1
Proceso 9
Proceso 12
Core 2
Core 3
Procesador 1
Proceso 0
Proceso 13
Core 0
Core 1
Proceso 14
Proceso 15
Core 2
Core 3
Procesador 0
Proceso 17
Proceso 20
Core 0
Core 1
Proceso 23
Proceso 26
Core 2
Core 3
Procesador 1
Proceso 18
Proceso 21
Core 0
Core 1
Proceso 24
Proceso 27
Core 2
Core 3
Procesador 0
Proceso 19
Proceso 22
Core 0
Core 1
Proceso 25
Proceso 28
Core 2
Core 3
Procesador 1
Proceso 16
Proceso 29
Core 0
Core 1
Proceso 30
Proceso 31
Core 2
Core 3
Procesador 0
Proceso 33
Proceso 36
Core 0
Core 1
Proceso 39
Proceso 42
Core 2
Core 3
Procesador 1
Proceso 34
Proceso 37
Core 0
Core 1
Proceso 40
Proceso 43
Core 2
Core 3
Procesador 0
Proceso 35
Proceso 38
Core 0
Core 1
Proceso 41
Proceso 44
Core 2
Core 3
Procesador 1
Proceso 32
Proceso 45
Core 0
Core 1
Proceso 46
Proceso 47
Core 2
Core 3
Procesador 0
Proceso 49
Proceso 52
Core 0
Core 1
Proceso 55
Proceso 58
Core 2
Core 3
Procesador 1
Proceso 50
Proceso 53
Core 0
Core 1
Proceso 56
Proceso 59
Core 2
Core 3
Procesador 0
Proceso 51
Proceso 54
Core 0
Core 1
Proceso 57
Proceso 60
Core 2
Core 3
Procesador 1
Proceso 48
Proceso 61
Core 0
Core 1
Proceso 62
Proceso 63
Core 2
Core 3

Fig. 9.3

9.3 Resultados comparados

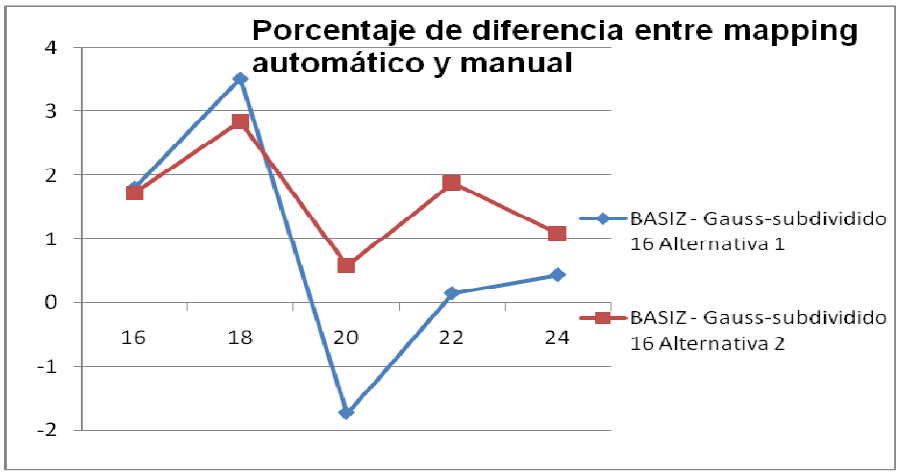

Fig. 9.4

10 Conclusiones y trabajo futuro

Tal como se explicó en la Sección 2, el objetivo de la Tesina es analizar e investigar de qué manera la asignación manual de procesos lógicos a procesadores físicos impacta sobre la *performance* de un algoritmo paralelo. Para ello se tomó como caso de estudio el problema *BASIZ* que permite desarrollar soluciones que utilizan como modelos de algoritmos paralelos el paralelismo funcional, el paralelismo de datos y una mezcla de ambos (híbrido).

La utilización de los modelos de paralelismo funcional, de datos y una combinación de los mismos (soluciones híbridas) permite evaluar el *mapping* manual en las diferentes alternativas posibles de implementación de un algoritmo paralelo. Esto posibilita hacer un análisis exhaustivo de cómo impacta la asignación manual de procesos en la gama posible de implementaciones. Es por este motivo que también se utilizan diferentes mecanismos de comunicación entre procesos, es decir, memoria compartida y pasaje de mensajes así como también una combinación de los mismos, es decir, soluciones híbridas.

Debido a que los tiempos de ejecución de los algoritmos implementados dependen del tamaño de la imagen que se utilice y no del contenido de la misma, el problema es escalable incrementando el número de imágenes.

Los resultados obtenidos muestran que el *speedup* del *Algoritmo BASIZ-Pipe-8* no mejora con respecto al *speedup* del *Algoritmo BASIZ-Pipe-5*. Esto se debe a que la descomposición funcional realizada está desbalanceada. Esto provoca que la fase más extensa en tiempo de ejecución (filtro gaussiano) no permita a los procesos que le siguen en el *pipe* trabajar de manera continua explotando el paralelismo, ya que como el tiempo de ejecución de los mismos es menor, deben pasar parte del tiempo esperando a que la fase más extensa termine para poder continuar procesando. Por este motivo, se implementó la solución BASIZ – Gauss subdividido, en la que la fase que en las soluciones anteriores provoca un cuello de botella (procesamiento del filtro gaussiano), es subdividida y procesada por mayor cantidad de procesos. Los resultados muestran una mejora importante tanto en el *speedup* como en la eficiencia del mismo si se lo compara con las dos soluciones anteriores.

Por otro lado, los resultados reflejan que en el algoritmo *BASIZ-MW* Replicado el *speedup* es mayor que en las soluciones *BASIZ-Pipe-5 y BASIZ-Pipe-8* dada la replicación que permite procesar más de una imagen simultáneamente. Además, puede observarse que el *mapping* manual de procesos mejora el *speedup* si se compara con el *mapping* realizado por el Sistema Operativo. Esto se debe a que la replicación, en la que lógicamente existen dos grupos de 4 procesos cada uno, permite mapear cada grupo a un procesador distinto de manera que los procesos de un mismo grupo comparten el mismo procesador físico y la misma *cache*. Sin embargo, si se la compara con la solución BASIZ – Gauss subdividido, su *speedup* y eficiencia no logran superar a esta última.

Asimismo, si se comparan los resultados obtenidos con las diferentes soluciones de paralelismo de datos, puede notarse que la solución *BASIZ* - Paralelismo Datos *OpenMPI* es la que mayor *speedup* y eficiencia alcanza. Mientras que si se comparan las dos restantes, la solución *BASIZ* - Paralelismo Datos *OpenMP* y *BASIZ* - Paralelismo Datos *OpenMP* 2 fases puede verse que la segunda alcanza, en promedio, mejor *speedup* y eficiencia que la primera. Sin embargo, si se comparan los porcentajes de diferencia entre la asignación manual y la llevada a cabo por el sistema operativo, las soluciones que utilizan

memoria compartida arrojan mejores resultados que la solución que utiliza pasaje de mensajes. Esto puede deberse a que al utilizar memoria compartida, el *mapping* manual fija la localidad de los hilos y de esta manera se evitan muchos cambios de contexto que provocan procesamiento innecesario.

También se han llevado a cabo pruebas para algoritmos que utilizan memoria compartida y pasaje de mensajes, tal es el caso de *BASIZ – Pipe*-5 *MPI* y *Pthreads* y *BASIZ – Pipe*-5 *MPI* y *OpenMP.* El *speedup* y la eficiencia obtenidos en ambas soluciones no mejoran con respecto a las demás soluciones sino todo lo contrario. Esto puede deberse a que los costos de sincronización en memoria compartida aumentan con respecto a los necesarios en pasaje de mensajes. Sin embargo, puede observarse que la solución *Pipe*-5 *MPI* y *OpenMP* arroja mejores tiempos de ejecución que la primera, si bien, a la hora de comparar el *mapping* manual, la solución que utiliza *Pthreads* arroja mejores resultados si se lo compara con el *mapping* del sistema operativo. Esto deberá ser tenido en cuenta cuando se quieran desarrollar aplicaciones que utilizan memoria compartida y se utilice el *mapping* manual para optimizar la *performance*.

Por último, las pruebas realizadas en el *cluster* de *multicore* arrojan resultados muy alentadores si se comparan las dos alternativas de *mapping* manual con la llevada a cabo por el sistema operativo. Esto permitirá seguir avanzando e investigando en esta arquitectura de tan reciente aparición en el procesamiento paralelo.

Como conclusión final, puede decirse que utilizando el *mapping* manual (que toma en cuenta las jerarquías de memoria) en soluciones que utilizan diferentes modelos de programación paralela y diferentes estrategias de descomposición, se mejora la *performance* alcanzable de los mismos si se los compara con la obtenida con el *mapping* del sistema operativo. Si bien las mejoras con el mapeo manual no son demasiado significativas cuantitativamente, los diferentes experimentos permitieron adquirir un *know-how* importante en cuanto a la asignación de procesos a núcleos. Esto es también escalable en arquitecturas de *cluster* de *multicore* tal como indican los resultados obtenidos.

A partir de estas conclusiones, como línea futura de investigación se puede pensar por un lado en investigar y analizar si la ganancia en *performance* obtenida por el *mapping* manual respecto al llevado a cabo por el sistema operativo se incrementa al aumentar el tamaño del problema.

Por otro lado, en implementar un *scheduler* que tenga en cuenta el tipo de aplicación, los recursos del sistema y las jerarquías de memoria existentes, de manera de ser parametrizable al momento de ejecutar una aplicación específica y de la cual el programador de aplicaciones se independice de la necesidad de implementar el *mapping* desde su aplicación, especialmente si se toma en cuenta que en un futuro, los procesadores *multicore* no serán todos de propósito general sino que cada núcleo tendrá funciones específicas, tales como multimedia, redes, entre otros.

11 Referencias

[1]AMD Multi-core White Paper (2009).

[2]Intel White Paper “La historia de los procesadores, desde ENIAC hasta Nehalem”.

[3]http://techresearch.intel.com/articles/Tera-Scale/1421.htm

[4]http://multicore.amd.com/la-es/AMD-Multi-Core/Multi-Core Advantages/Triple-Core-Advantages.aspx

[5] “AMD Opteron Processor Update” (Junio 2008).

[6]*Siddha S. “*Multi-core and Linux* Kernel”. *Intel Open Source Technology Center*.

[7]AMD White Paper “Exploiting Multi-Core Processors in Windows Vista”.

[8]Microsoft Development Network “Multi-Core Support in Windows 7”

[9]Dongarra J. , Foster I., Fox G., Gropp W., Kennedy K., Torzcon L., White A. “Sourcebook of Parallel computing”. Morgan Kaufmann Publishers – ISBN 1 55860 871 0 (Capítulo 3).

[10]Grama A., Gupta A., Karpyis G., Kumar V. “Introduction to Parallel Computing”. Pearson – Addison Wesley – ISBN: 0 201 64865 2. Segunda Edición (Capítulo 3).

[11]Kumar V., Gupta A., “Analyzing Scalability of Parallel Algorithms and Architectures”. Journal of Parallel and Distributed Computing. Vol 22, nro 1.Pags 60-79. 1994.

[12]Leopold C., “Parallel and Distributed Computing. A Survey of Models, Paradigms and Approaches”. Wiley,2001. ISBN: 0471358312 (Capítulos 1, 2 y 3).

[13]Andrews G. “Foundations of Multithreaded, Parallel and Distributed Programming” Addison Wesley Higher Education 2000. ISBN-13: 9780201357523 .

[14]De Giusti L. “Mapping sobre Arquitecturas Heterogéneas”. Tesis Doctoral, Universidad Nacional de La Plata (2008).

[15]Amdahl, G.M. “Validity of the single-processor approach to achieving large scale computing capabilities”. In *AFIPS Conference Proceedings* vol. 30 (Atlantic City, N.J., Apr. 18-20). AFIPS Press, Reston, Va., 1967, pp. 483-485

[16]Gustafson, J.L. “Reevaluating Amdahl's Law”. CACM, 31(5), 1988. pp. 532-533.

[17]Zoltan J., Kacsuk P., Kranzlmuller D., “Distributed and Parallel Systems: Cluster and Grid Computing”. (The International Series in Engineering and Computer Science). Springer; 1st edition, 2004.

[18]https://computing.llnl.gov/tutorials/pthreads

[19]https://computing.llnl.gov/tutorials/openMP

[20]http://www.open-mpi.org

[21]Ghuloum A., Sprangle E., Fang J., Wu G., Zhou X. “Ct: A Flexible Parallel Programming Model for Tera-scale Architectures” (2007).

[22]Roig C. “Algoritmos de asignación basados en un nuevo modelo de representación de programas paralelos” Tésis Doctoral – 2005 Unidad Autónoma de Barcelona –España.

[23]Roig C., Ripoll A., Borrás J., Luque E. “Efficient Mapping for Message-Passing.Applications Using the TTIG Model: A Case Study in Image Processing.”

[24]Roig C., Ripoll A., Senar M. A., Guirado F., Luque E. “A New Model for Static Mapping of Parallel Applications with Task and Data Parallelism”.

Procesador de texto utilizado: *Microsoft Word* 2007.